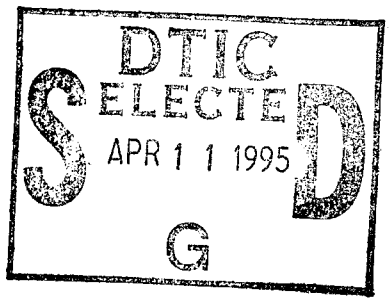
ARMY RESEARCH LABORATORY

Automated Translation of Bit-Oriented Messages (BOMs) Into Data Kernel Representations (DKR)

George W. Hartwig, Jr.

ARL-TR-728 April 1995



NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute endorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1995	Final, Feb 91–May 92

4. TITLE AND SUBTITLE
Automated Translation of Bit Oriented Messages (BOMs) into Data Kernel Representation (DKR)

5. FUNDING NUMBERS
4B592-50234-3004
CC: 4B3400

6. AUTHOR(S)
George W. Hartwig, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-CC
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-728

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A methodology along with a prototype implementation for the automated translation of Bit-Oriented Messages (BOMs) into Data Kernel Representation (DKR) is presented. DKRs, being extensions of Information Distribution Technology (IDT) fact types, are collections of information that define battlefield events or actions in a generic, computer oriented fashion. Since the translation of BOMs into DKRs is designed to be bi-directional, a collection of DKRs, could serve as a meta-language for the distribution of tactical information across battlefield functional areas. The prototype implementation, written in the C programming language, is described and example template files for representative Combat Vehicle Command and Control (CVC2) messages and TACFIRE messages are shown.

14. SUBJECT TERMS	15. NUMBER OF PAGES
tactical communications, digital communications, message processing	135
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK.

LIST OF FIGURES

INTENTIONALLY LEFT BLANK.

1. INTRODUCTION

As the Army enters an era of reduced budgets and downsizing, the need to capitalize on technology as a force multiplier increases. By the use of high-technology sensors, sophisticated command and control, accurate long-range weapons, and highly trained soldiers, the modern Army hopes to offset the reductions of men and weapons with highly maneuverable combined arms teams. Perhaps the most difficult factor in this scenario is maintaining the communication links necessary for command and control and dissemination of time-critical intelligence to the soldier. The digital communication systems of the various Army Battlefield Functional Areas (BFA)—Maneuver Control, Air Defense, Fire Support, Combat Service Support, and Intelligence/Electronic warfare—were developed independently to meet specific needs within each BFA and, as a consequence, did not address the problem of interfacing with other BFA protocols. Any required information exchange between BFAs takes place via voice radio channels or at battalion or higher echelons via "swivel chair" interfaces in which a soldier reads from one device and types into another. Combined with the narrow bandwidth communications available with Combat Net Radios (CNR), the problem of coordinating task force organizations, assimilating sensor information, and maintaining positional awareness has become increasingly difficult. Complicating this already difficult situation is the fact that these task forces may be multinational in composition, thereby introducing a language problem into the voice communications that often provide the only BFA coordination available today at the fighting echelons.

To address these problems, the Advanced Computational and Information Sciences Directorate (ACISD) of the U.S. Army Research Laboratory (ARL), formerly the Ballistic Research Laboratory (BRL), has and continues to develop an experimental Information Distribution Technology (IDT) to explore innovative concepts to facilitate the exchange of tactical information over low bandwidth communications channels. This work was begun as part of the Army DARPA Distributed Communication and Processing Experiment (Chamberlain 1986) and continued as part of the Smart Weapons Systems (SWS) U.S. Army Laboratory Command (LABCOM) Cooperative Program (Rogers 1991). This project was supported as part of the U.S. Army Communication and Electronics Command (CECOM) Directorate for C3 System's Low Echelon Command and Control (LEC2) Program.

2. STRATEGY AND METHODOLOGY

The approach selected for IDT prototype work was to separate the information problem into two parts: one of information presentation and the other of information distribution. Previous work has concentrated on the problem of information distribution. A data dictionary has been created that is composed of Data Kernel Representations (DKRs). To perform the distribution and storage of the DKRs, the IDT Distributed FactBase (DFB) was utilized. When combined with application programs that provide for battlefield management and interfaces with the combat soldier, the DFB not only provides storage for DKRs in a memory-resident database but also allows for the automatic distribution of information via predefined rules and standing queries or "triggers."

2.1 Program Functionality. The purpose of this project was to successfully translate the bit-oriented messages (BOM) of one BFA into corresponding BOMs of another BFA. Given an ongoing familiarity with fixed format TACFIRE messages and documentation on a proposed Combat Vehicle Command and Control (CVC2) protocol provided by the CECOM Center for C3 Systems, it was resolved to develop an experimental methodology for translating back and forth between TACFIRE and CVC2 messages.

Since the algorithm would eventually be required to interpret many different BOM protocols, to create individual interfaces between each pair of protocols was deemed to be unfeasible. Based on our background of IDT work, a meta-language consisting of DKRs was determined to be a reasonable interface medium; so the translation program (trans_p) was designed to translate BFA BOMs into appropriate DKRs and generate BFA BOMs from DKRs. To accomplish this, template files are created which define the locations and sizes of the fields in the various BOMs and describe the conversions necessary to convert back and forth between BOM format and DKR format. Associated with each set of protocol template files is a library (which must be linked with trans_p before execution) of "helper" routines, which perform computations too difficult to specify in the template file, and auxiliary functions such as communication with protocol-specific hardware such as modems, etc.

Currently, the issue of to whom messages transversing BFA boundaries should be sent is not contained within the DKRs or the template files. Since such information will depend on the scenario in which the translation program is operating, these issues are considered in two separate places. Addressing on a particular BFA net will be contained in an address file to be processed by the translation program upon

startup. The issue of inter-BFA communications will be sidestepped entirely by leaving it up to the DFB's rule mechanism (Hartwig, unpublished).

2.2 Data Kernel Representations. The DKRs are collections of information that define battlefield events or actions in a generic fashion so as to span BFAs. The DKRs shown below represent the latest version in the continuing process of IDT fact types development (Chamberlain 1986). The DKRs provide a common basis for information distribution that has been shown to be sufficient to represent fire support activities (Rogers 1991) and has been recently expanded to include air defense requirements (Smith 1992). At this point in their continuing evolution, there are 14 DKRs for the purpose of combat information exchange. They are as follows:

g_type	The DKR version of the operations order and as such is continually revised.
line	A fact representing lines. Used primarily in conjunction with display programs to display borders, routes, and areas.
grid	Used to locate a point. Contains easting, northing, a time, and 64 bits worth of terrain data. Format may change in the near future to allow a more universal representation to be used.
equip_attr	Used to specify maximum or effective ranges for referenced equipment.
status	Used to specify quantities. Composed of an integer field and an associated DKR id number.
commo	Used to support inter-application communications. Application-specific DKR.
ammo	Round, fuze, and propellant information is contained in this DKR.
equip	Describes all kinds of miscellaneous equipment and weapons.
veh	Contains important information about each vehicle in the table of organization and equipment (TO&E) including range, speed, onboard equipment, etc.
target	Links sensing DKRs to attacking units. This target definition from the fire-support BFA.
sensing	Contains intelligence information about the battlefield. In particular, sightings of enemy activity are represented.

what_if	Allows for planning to occur by representing "what if" situations by providing alternative fields for referenced DKRs.
org_type	A reference DKR providing the TO&E information for U.S. Army units.
unit_type	Describes an actual fielded unit including all status, position, and activity information.

Note that application programs are free to define DKRs for their own use. However, these "local" DKRs may not be sent to remote DFBs without prior coordination. The translation program creates one of these "local" DKRs called CUR_MSG for each incoming BFA message, thus forming a history of incoming messages. This DKR contains information about received messages, including the following:

msg_prot	Type of protocol, e.g., CVC2.
msg_type	Type of message within the protocol.
msg_wilco	1 => wilco required for this message.
msg_src	The source of this message; format depends on the protocol of the incoming message.
msg_dst	The intended destination of this message.
msg_trynum	The retry number for this message.
msg_auth	Special authorization for this message.
msg_chan	Channel this message was received on, if known.
msg_time	Time this message was received.
msg_priority	Message priority.
msg_len	Length of this message (in bytes).
msg_ref_facttype	Name of referenced fact type.
msg_ref_factid	Referenced fact.

The approach selected to solve the problem of translating between BFA digital communication protocols was to define a meta-language composed of DKRs and create a computer program to perform

the bidirectional translation between BFA protocols and DKRs. To gain a preliminary understanding of the problems involved in creating such an interface, translation between CVC2 and TACFIRE was considered. CVC2 message formats are BOMs, and for the most part, each field refers to a table of predefined meanings. TACFIRE messages are similar, but the fields are composed of characters from a limited set. A few fields are multiple characters wide, and they are specially coded. However, to be considered successful, the selected methodology must be general enough to allow easy expansion into the other BFA protocols as shown in Figure 1.

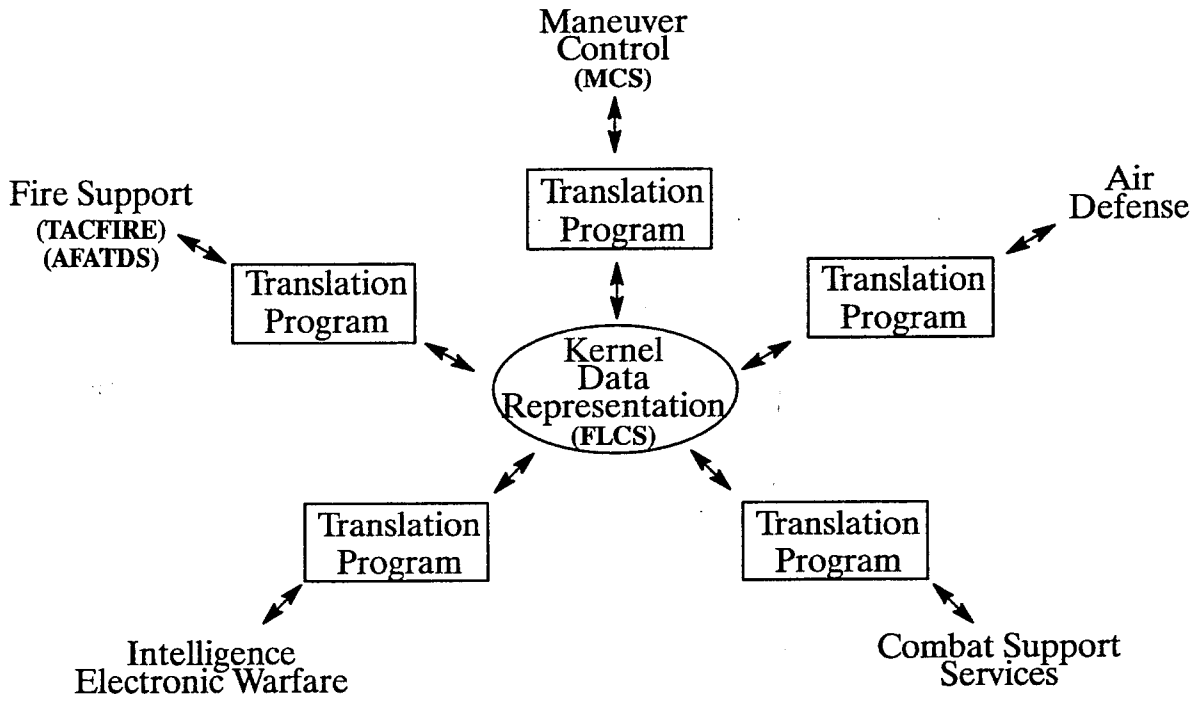


Figure 1. Inter-BFA operability.

This report will present a methodology for data-driven interpretation of BOMs. The procedure is general enough that by the creation of additional descriptive files for other BOM protocols, the same program may be used to translate the messages of those protocols into DKRs. The prototype implementation in the C programming language is also discussed.

The flow of data through the translation program is shown in Figure 2.

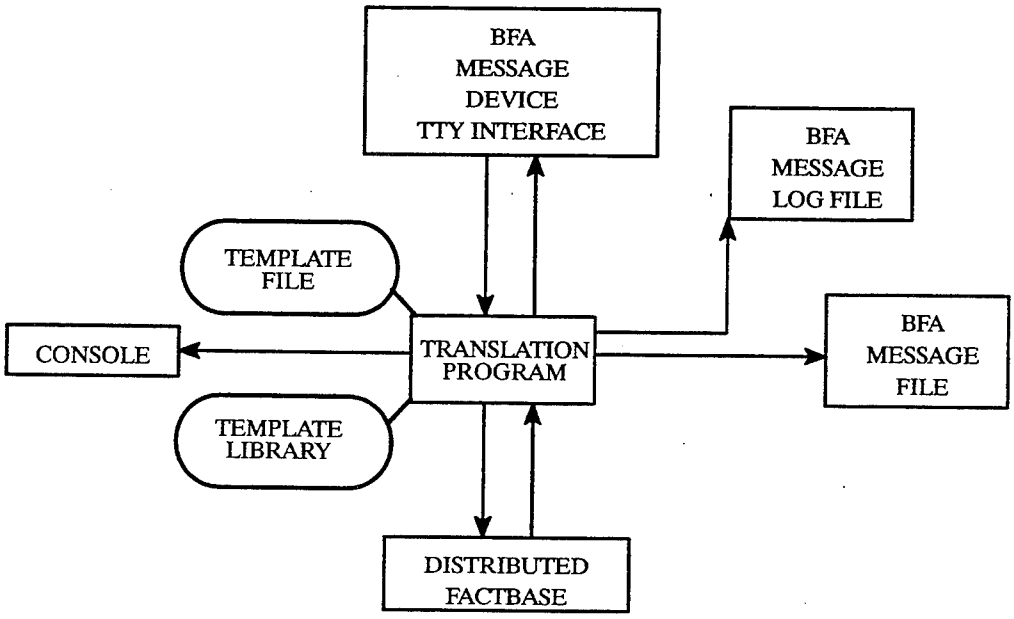


Figure 2. Message flow.

The primary routes of message flow are from the BFA TTY* interface to the DFB and from the BFA message file to the DFB. Information from the DFB flows to the BFA TTY interface and, if requested, to the BFA log file. BFA message files and BFA log files are of the same format, so the translation program can be used to interpret their own log files.

2.3 Template Files. The operation of the translation program is completely governed by the template files. The translation program, when combined with the template files and associated library functions for a particular protocol, provides an interface between that protocol and a DFB node.

* The physical interface to the protocol-specific hardware is through the serial RS232 ports of the host computer, hence the name TTY interface.

The template file organization begins with a protocol file. This file associates the names of various messages within the protocol and the respective template files to interpret them. Also specified within the protocol file is the name of a function that is used to interface with protocol-specific hardware. The protocol file used with the demonstration set of TACFIRE messages is as follows:

```
tacfire.protocol    5 tf_comm
# filename          Message type
tacfire_header      "TACFIRE HEADER"
tacfire_frgrid      "TACFIRE FR GRID"
tacfire_atigrid     "TACFIRE ATI GRID"
tacfire_freetext    "TACFIRE FREETEXT"
tacfire_acknak      "TACFIRE ACKNAK"
```

The first element is just the name of the protocol being described and serves as a check. The second element is the number of templates, including the header template, named in this file. The third and final element is the name of a library function to handle handshaking with BFA-specific hardware. Lines beginning with '#' characters are comments. Subsequent lines relate protocol messages and their corresponding template files. The first entry is the template file name, and the second entry is the name of the BFA message.

Each message template consists of three parts. The first is simply the message name as specified in the protocol file that is used as a check that the proper file is referenced. The second part is a listing of DKR types that are created when an incoming BFA message is received. Under each DKR type, fields are listed that are to be added to the created DKR, but are not found in the BFA message. These are typically control fields that do not appear in the incoming message. For instance, the DKR section for a CVC2 CALL FOR FIRE message is as follows:

```
sensing
        mode = 21

target
        src = @sensing
        time = 0
END
```

Two DKRs will be created when a CALL FOR FIRE is received. The "sensing" DKR will have the field mode filled with the value 21, and the "target" DKR will have the field time set to 0 and the field src filled with the DKR identifier of the "sensing" DKR; thus the target field src will be a pointer to the "sensing" DKR. The keyword END terminates this section.

The final and by far the largest section is that containing the actual message field descriptors. These descriptors not only describe how to interpret an incoming BFA message, but how to create a BFA message as well. When this program was being created, the first protocol examined was CVC2. In this protocol, most fields contain small integers that are then used to look up the appropriate value in a table. These are better known as enumerated fields. Occasionally numeric fields also appear in which the numerical value is interpreted literally. Some of the numeric fields are special in that they contain control information such as the number of instances of a repeated field. These original field types were called ENUM, NUM, and RPT_I. Later, as the fixed format TACFIRE protocol was considered, it became clear that additional field types would be needed. Since fixed format TACFIRE is really character oriented with a limited character set, it became necessary to add the CHAR and TBL field types. Therefore, the current recognized fields are as follows:

ENUM	Number from field is used as an index into a table to get the actual meaning.
RPT_I	Specifies the number of times to repeat other fields in this message.
NUM	Nil - numeric interpretation.
CHAR	Interpreted as a 7-bit ASCII character.
TBL	Uses the message value as an index into a table to obtain the true meaning. Different from the enum type in that field values may not be sequential.
SPARE	Unused field indicator.

The individual fields in each message are defined by entries in the template file. The form of these entries depends on the type of field being described. All fields have the same first line. The four entries on this line are as follows:

FIRST LINE - REQUIRED FOR ALL FIELDS

position	Bit position in the message at which this field begins. Assumes that the message is a bit stream numbered starting at 0.
length	Bit length of this field. Includes the initial bit position.
type	Type of this field, i.e., ENUM, TBL, etc.
name	Message name of this field.

The second line of the field entry is required for all data fields. This line specifies where incoming message data are to be stored, where outgoing data are to be retrieved from, and how they are to be processed.

SECOND LINE - REQUIRED FOR ALL DATA FIELDS

dkr_type	The DKR to which this data field corresponds. The dkr_field when combined with this field completely specifies where the information in this message field is to be stored.
dkr_field	The data kernel element in which to store this message field.
dkr_source	Field specifying the DKR field that contains the information to enter in this field when it becomes time to generate a BFA message.
dkr_default	Value used in the outgoing BFA message when all other sources fail.
trans_func	The name of the function required to process the message field into a format compatible with the DKR.

The dkr_type is typically just the name of a DKR, but two other options are available. If the message field has no corresponding DKR representation, then the keyword NONE may be inserted. If this option is used, the dkr_field must also have the value NONE inserted. The second option was created to accommodate the fact the CVC2 protocol places the sender's location in the message header. The construct "$.cur_msg.msg_src" allows the template to reference values in the cur_msg DKR. In this case, the specific DKR representing the sending unit is specified.

The dkr_source field is the source of information to be placed in this field in an outgoing message. Since all outgoing messages begin with a trigger notification, all referencing must begin from the triggering DKR. An entry of the form "$.fooy" would cause the value for the field called "fooy" in the

triggering DKR to be used as the field value in the BFA message. Syntax of the form "$.name1.name2.fooy," where all the names except the last are references to DKRs, would cause a path to be traced through the referenced DKRs to the field "fooy." Thus beginning with the DKR that caused the trigger to fire, we may pinpoint specific fields in specific facts for information to insert into BFA messages. The keyword NONE may be used when the information to fill this field cannot be found as a DKR entry.

The dkr_default entry is, as the name implies, the value used to fill in a BFA message field when other options have been exhausted. This value should be human intelligible, and it is generally the value found in the DKR. If the value is more than a single word, it is placed within double quotes. To accommodate the sender address found in the CVC2 header, the construct "$.addr.me.east" is allowed. This causes the program to look in the address file information under the trigger name "me," look up the appropriate DKR, and use the value found in the field "east."

The last element, trans_func, is a protocol-specific library function. These functions are basically helpers, enabling more detailed data processing than is available to the general field processing software. These functions have to operate bidirectionally, i.e., forward (BFA->DKR) and reverse (DKR->BFA). They are often used where it is necessary to alter the data as interpreted or where several fields on one side coalesce into a single field on the other. If a translation function is present, it is always called, and it is expected to perform all translation duties.

ENUM FIELD TYPES

SUBSEQUENT LINES

table — The table of strings representing appropriate entries for this field. The value found in the message is used as an index to this table.

The above explanation is straightforward. The table should be terminated with the keyword END.

TBL FIELD TYPES

SUBSEQUENT LINES

table — The table consisting of index values followed by the DKR value.

The table is composed of lines, each of which consists of an index followed by a value, typically a string. If the index is inclosed within single quotes, then it is interpreted as an ASCII character and the numeric value of that ASCII character is inserted into the BFA message. If on the other hand the index is not inclosed within quotes, then the index is interpreted directly as a number and the corresponding value inserted into the outgoing message. See examples in Appendix A-1. The table should terminate with the keyword END.

RPT_I FIELD TYPES

OPTIONAL SECOND LINE

number	The number of fields to be repeated.
func_name	The name of a helper function if needed.
field_name	The name of the first field to be repeated.

CHAR FIELD TYPES

NUM FIELD TYPES

These field types do not have any special lines.

Following the above lines, each field description has a set of lines that determine which field is to be processed next. This procedure was adopted to handle the case of optional fields or cases where the field size depends on the value of a previous field. Examples are found in Appendix A-2. The syntax for these control lines is:

reference field reference value new_field

where reference field is the name of a previously processed field or one of the keywords ABSOLUTE or SELF. Lines with the ABSOLUTE keyword have no reference value since the field named in new_field is processed next absolutely with no dependence on prior fields. The keyword SELF implies that subsequent processing depends on the value of the field currently being processed. The reference value is the value that must be matched. This value must not be the index value but rather the human readable or DKR value. The keyword ANY may be used in this field. In this case, any value will cause a match. New_field is the name of the next field to be processed. The keyword TEMPLATE_END denotes the end of processing for this message. Any template entry in this section that consists of more than one word

must be enclosed in double quotes. The order of lines is important since the first match will determine subsequent processing. An unsuccessful match will cause problems and probably cause catastrophic failure and program termination. This section terminates with the keyword END.

2.4 Address Files. The address file is used to specify communications on the BFA side of the translation program. Since all outgoing BFA messages are the result of information from the DFB, the DFB "trigger" mechanism is used to key BFA message creation. The "trigger" mechanism allows an application program to tell the DFB what information it is interested in. A trigger request consists of an identifier, a DKR type, and a criterion. When information enters the DFB, altering the named DKR type and satisfying the specified criterion, a notification is sent to the application program that stated that trigger to the DFB. The translation program upon receipt of the trigger notification retrieves the triggering DKR and generates the specified BFA message.

All information required to generate the outgoing BFA message is contained in the address file and the templates for the particular BFA under consideration. The format for the address file is as follows:

trigger_name; DKR_type; expression; msg_type; unit_id; unit_address

where:

trigger_name	is an identifying name for this trigger. It is used by the DFB to identify the trigger to the application program.
DKR_type	is the DKR type this trigger applies to. Whenever a DKR of this type is created or modified, the following expression will be tested.
expression	is the triggering expression. Whenever this expression is satisfied, a notification is sent to the stating application program. This expression is copied directly into the trigger statement sent to the DFB.
msg_type	is the type of message that will be sent when the preceding expression is satisfied.
unit_id	is the identification number for a particular unit. This number is found in the DKR unit_type.
unit_address	is the unit address as determined by the BFA protocol being used.

An example address file is shown here:

```
# This is a test file for the CVC2 node.  Addresses reflect the CVC2
# addressing scheme.
#
# Trig_handle  Fact_type  Trig_expr  Msg_type  Unit_name  Unit_addr
#
# 2-11 BN TOC (OPNS)
me NONE "($.idnum == \"B3221121\")" NONE  B3221121 0xd180
#
# B/1-440 AIR DEFENSE BN
#
trig_1  sensing ($.mode==22)  "CVC2 SPOT REPORT" B3221211  0x1300
#
```

Lines beginning with a '#' character are comments and may contain any useful information. The first data line is a special line and identifies the translation program's address on the BFA network. On this line, only the trigger handle, unit_id, and unit_address are used. No trigger is built for this line. The expression on this line is simply to serve as an example and serves no other purpose than a place holder to satisfy the parser. The unit_addr field is copied directly in the source field of outgoing messages and therefore, is dependent on the protocol. In this example, the unit_addr is binary and maps into several fields in CVC2 header. Note that the leading "0x" in the unit_addr means that this is a hexadecimal representation of the binary address.

In this example, when the DFB receives a sensing DKR update that has the mode variable set to 22, a notification will be sent to the translation program which will then generate a "CVC2 SPOT REPORT" that is then sent to the B/1-440 AIR DEFENSE BN at CVC2 address 0x1300.

2.5 Protocol Specific Library. Occasionally it is necessary to perform computations to effect the translation from DKR to BOM field or vice versa. It may also be necessary to collect several message fields into a single DKR field or vice versa. Protocol-specific hardware hand shaking may also be required. The template files are not well suited to these tasks so each BFA template has associated with it a library of functions that are linked to the translation program to form a specific BFA translator.

These functions are named in the template file and called by the translation program via a switch routine that searches an array of structures that relate the function names to pointers to the functions. This structure is shown below.

```
struct FUNC_TAB {
char *func_name;        /* Name of a function              */
char *(*func_ptr)();    /* Pointer to named function       */
                        /* This construct may cause        */
                        /* some portability problems in    */
                        /* moving to alternative machines. */
};
```

First in the library is a communications function that is referenced in the "protocol" file. The argument list contains the following.

opt	One of INIT, IN, or OUT and specifies the action to be performed.
tty_file	The name of the input/output device to be opened as the BFA interface.
message	A pointer to a pointer to an IN_MSG structure.

When the INIT option is specified, the communication function must open the TTY device as specified by the second argument "tty_file" and set parameters as needed. It must also do any initializing of attached hardware. When called with the IN option, this function must assemble a complete BFA message from the TTY interface and store it in a dynamically allocated IN_MSG structure. This task is complicated by the fact that the message may be spread in time since tactical communication media tend to be slower devices than the associated digital computers. So the IN portion of the communications function must hold partial messages, strip off hardware-required headers and trailers, and return complete messages as they arrive. If the OUT option is selected, a BFA message is taken from the IN_MSG structure, appropriate headers and trailers attached, and the message sent to the TTY interface. Note that this function may or may not have to generate and process acknowledgments (ACK) to be sent to BFA correspondents.

The rest of the functions found in the library are called translation functions and perform computations necessary for translation of field information or for gathering information from multiple sources. They may also spread information to multiple destination fields. Since each translation program is designed to be bidirectional, each translation function may be called with either the FORWARD or REVERSE option set. The FORWARD direction is from BFA to DKR and REVERSE is the opposite. These function are discussed further in the implementation section below. Appendix B presents a typical library translation function.

As a simple example of the field translation process, assume that we wish to relate fields which specify a temperature. The incoming message has a field called "amb temp" which is defined as 8 bits starting in bit position 45 in units of degrees Fahrenheit. We will store this quantity in a DKR as a 32-bit integer in degrees centigrade. Then we will create a target message field that is a 3-bit quantity beginning in bit position 25 with the following subjective meanings:

000 dangerously cold
001 pretty cold
010 cold
011 nice
100 moderately warm
101 warm
110 hot
111 dangerously hot.

The information we need to interpret this incoming field may be summarized as:

Line 1

Start Position	Field Width	Field Type	Field Name
45	7	NUM	"ambtmp"

Line 2

DKR Facttype	DKR Factfield	DKR Source	DKR Default	Translation Function
met	temp	temp	"nice"	f2c

This tells us that the field of interest may be found in message bit position 45, and this bit along with the next seven bits represent a number. This number then may be interpreted using the knowledge found in a subroutine called "f2c" to arrive at the DKR representation, i.e., we must apply the formula

$$C = (5/9)*(F - 32) .$$

Then the resulting temperature is stored in the DKR via the command

state met { ambtemp = C };

Finally, to produce the corresponding field in the outgoing message, we need to know the following:

Line 1

Start Position	Field Width	Field Type	Field Name
25	3	ENUM	"ambtmp"

Line 2

DKR Facttype	DKR Factfield	DKR Source	DKR Default	Translation Function
met	temp	temp	"nice"	c2enum

Subsequent lines

very cold
pretty cold
cold
nippy
cool
warm
hot
steamy.

(Note: These examples only contain the lines necessary to describe the field. Control entries that would usually follow the ENUM table are missing.)

The ENUM field type designator tells us that this field is an index into a limited set of values. Based on the temperature stored in the DKR, we pick the appropriate index and store it. However, the question of what is the correct index is unanswered. Is 20° C "cold," "nice," or "moderately warm?" To solve this dilemma, the translation function, "c2enum," must contain some rules, or heuristics, to help us with the translation. Table 1 provides us with a method of encapsulating these heuristics:

Table 1. Example Heuristics

Temperature (°C)	Nomenclature	Index
–21 or less	dangerously cold	000
–20 to –11	pretty cold	001
–10 to –1	cold	010
0 to 9	nice	011
10 to 19	moderately warm	100
20 to 29	warm	101
30 to 39	hot	110
40 or greater	dangerously hot	111

So, to fill in the outgoing message field, we take the DKR entry and choose the correct index.

A message coming the other direction is treated in the same manner except that the algorithm is run in reverse. Since we store a number in the DKR, we must have a heuristic to allow the picking of a specific temperature from the indicated range. For example, if the received index is 100, we must select a single number from the indicated range, "10 to 19." A possible choice would be 15.

3. IMPLEMENTATION OVERVIEW

In this section, an overview of program execution is provided. The intention is to demonstrate the many steps that must be taken to successfully perform the bidirectional translation of DKRs to BFA BOMs. The distinct tasks of program initialization, template processing, address file processing, message processing, DFB interaction, and TTY interaction will all be presented. Appendix C contains the manual pages for the prototype implementation TRANS_P.

3.1 Program Initialization. When starting, TRANS_P checks for a minimum set of command line arguments, then the command line arguments are processed. Valid command line arguments include:

-p protocol_file	A file containing the protocol top-level template. It is a required argument.
-l	Load template files only. Used to verify that protocol template files are at least grossly correct.
-a address_file	Address file name. Contains the addresses of other units on this net and criteria for sending them messages.
-i message_file	A file of BOMs to be interpreted and processed as specified by other arguments.
-t tty	The tty port to which actual interface hardware is connected. This port will be monitored for incoming BOMs, and appropriate outgoing messages will be sent to this port.
-T time	A time delay between processing messages found in the message_file. Used when the translation program is serving as a scenario driver.
-h hostname	A flag used to specify the host running the DFB when the 'd' flag is present. The default is the local host.
-d	Connect to a DFB. When a BOM is received, fact updates will be sent to this DFB.
-r	Read to run flag. This flag indicates that a character must be typed at the keyboard before a message from the message_file is processed. Allows finer control than with the 'T' flag.
-D	Turn on debugging prints for lots of reading enjoyment.
-c	Print on the console human readable versions of each BOM processed.
-L log_file	Log each BOM message received.

Finally, the line parser is initialized. This parser divides each line into a set of tokens. Strings contained within double quote (") characters are maintained as single units. Recognized delimiters include backslash, space, newline, tab, double quotes, and zero.

The protocol templates are read in and stored. This procedure is detailed in the Template Processing Section. If the protocol file cannot be found or opened, the program terminates. If the "–l" flag is

present, execution terminates after attempting to load the specified protocol. This feature is useful in testing a new template file without worrying about having a DFB running or other communication options.

If the "-d" flag is set, TRANS_P then attempts to connect to a DFB. If a remote host has been specified through the use of the "-h" option, an attempt is made to create a connection to that host; otherwise a DFB running on the local host is searched for. All connections are made through the services of the Package (PKG) Protocol. This protocol is described in Appendix D of this report.

The address file is mandatory for successful operation of the translation program. Only template loading may be performed without the address file being present. As this file is processed, DKR_IDs corresponding to referenced units are retrieved from the DFB and "triggers" are set. "Triggers" are discussed more fully in the section on interaction with the DFB.

If a log file has been called for, it is opened at this point. Log files include both incoming and outgoing BFA messages. The format is 4 bytes of message byte count (rounding up) followed by the BOM. Since these are BOMs, any character may be present in the message, including newline and NULL. A log file may serve as an input file to another translation program.

If the "-t" flag is present and a protocol communication function is present, then at this time a call is made to the communication function with the INITIALIZE command specified. This function must then open and set appropriate parameters on TTY lines and perform any other initialization required for protocol-specific hardware interfacing.

Now program execution has reached the stage where message translation may occur. If the "-i" flag was used to name a message input file, that file is now opened, and as long as messages are present, the following actions occur: (1) the message is translated, (2) if requested, the translated message is printed on the console, (3) if a DFB is connected, DFB commands are created and sent to the DFB, and finally, (4) a check is made for any trigger messages that may have been sent by the DFB.

After the message input file has been processed or if no message input file was specified, the translation program falls into a loop waiting for incoming information from either the BFA interface or the DFB. The UNIX (AT&T trademark) system call "select" is used to perform the waiting function. If data is determined to be present on the BFA interface or the DFB connection, then appropriate processing

is done. This includes checking for a terminate request from the console, processing trigger messages from the DFB, and assembling and translating BFA messages from the TTY interface. In the case of a trigger message, a BFA message is constructed and sent to the addressee via the TTY interface. If a BFA message has arrived, DFB commands are constructed as specified by the template files and the message sent to the DFB.

3.2 Template Processing. In an earlier section, the general structure of the template files was presented. In this section, the details of run time storage are given. Since the processing of the template files represents a relatively large amount of computation, it is necessary to load the information from the template files into memory for ease of access and speed of translation.

The top level for each protocol is represented by a PROTOCOL structure. This structure is statically declared and acts as the anchor for the stored template. The PROTOCOL structure is defined as follows:

```
struct PROTOCOL {

    char *p_name;                   /* Protocol name                          */
    int p_num_msgs;                 /* Number of message templates            */
    char *p_comm;                   /* Name of a function to handle           */
                                    /* commo on the tty port to the bfa       */
    struct MESSAGE_HEAD **p_loms;   /* Array of pointers to message templates */
};
```

The PROTOCOL structure contains a pointer to an array of MESSAGE_HEAD structures that in turn refer to the individual field templates. This is represented schematically in Figure 3.

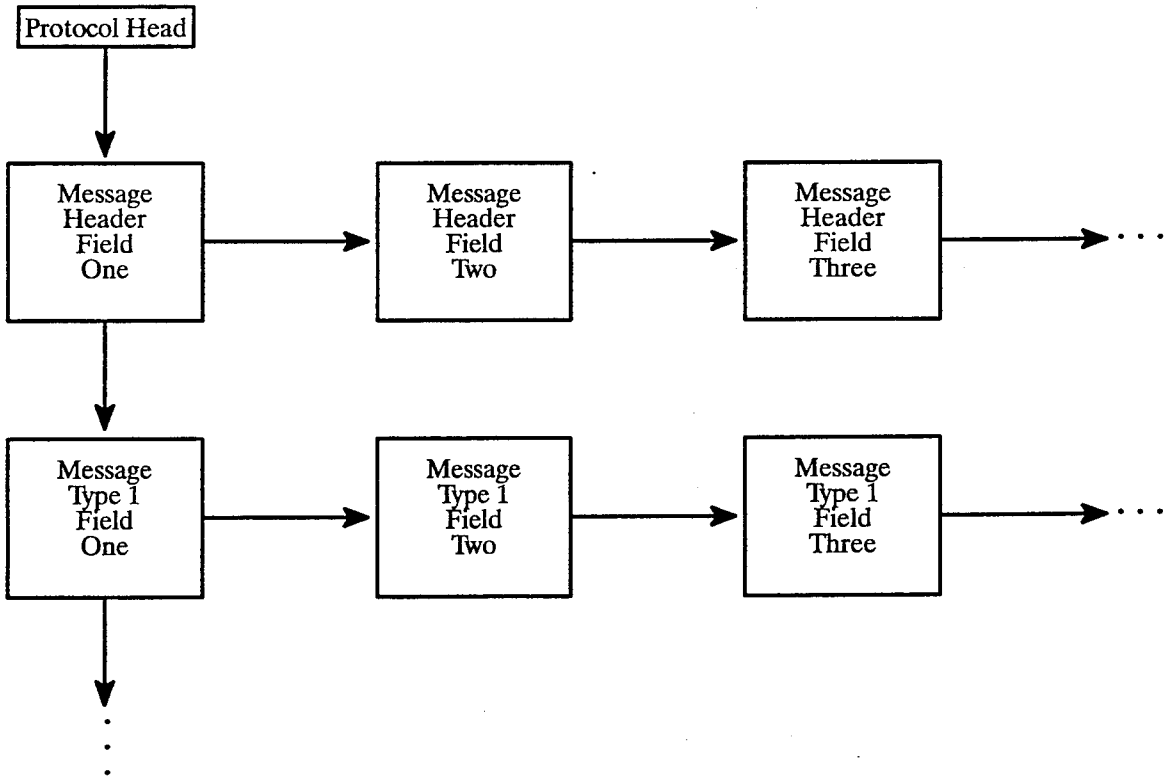


Figure 3. In memory template organization.

The corresponding data structures for the MESSAGE_HEAD and FIELD structures are as follows:

```
struct MESSAGE_HEAD {

    char *msg_title;              /* Name of the message.              */
    struct REF_DKRS *ref_dkrs;    /* Names of any DKRs referenced      */
                                  /* by the message template.          */
    struct FIELD *msg_fields;     /* Decoding information              */
                                  /* for the fields.                   */
};

    struct FIELD {

    int  f_position;              /* Bit position in message bit stream. */
    int  f_length;                /* Bit width of field.               */
    int  f_type;                  /* NUM, ENUM, RPT_I, etc.            */
    char *f_name;                 /* Name of this field must be unique. */
```

```
    char *f_trans_func;             /* A function used to help with the    */
                                    /* interpretation of the binary value. */
    char **f_etable;                /* A table of enumerated values.       */
    struct TABLE *f_table;          /* A limited table of indices and      */
                                    /* values.                             */
    int r_number;                   /* The number of fields to repeat.     */
    char *r_field;                  /* Name of field to start repeating    */
    struct FIELD *r_start;          /* Pointer to FIELD structure where    */
                                    /* repeating starts.                   */
    char *f_dkr_type;               /* A multipurpose field for            */
                                    /* specifying to which DKR element this*/
                                    /* message field corresponds. If       */
                                    /* the first character is a letter     */
                                    /* from [a-z], this is a DKR-type name */
                                    /* and will result in a new fact being */
                                    /* stated to the DFB. If the first     */
                                    /* character is '@' and the second     */
                                    /* character is '"', then this         */
                                    /* represents a specific fact and the  */
                                    /* appropriate field in that fact will */
                                    /* be updated. The final option is if  */
                                    /* the first character is '@' followed */
                                    /* by either msg_src or msg_dst, then  */
                                    /* we will look to the current message */
                                    /* header for appropriate references.  */
    char *f_dkr_field;              /* Specifies the specific field in a   */
                                    /* DKR to be either stated or updated  */

    char *f_dkr_source;             /* When it becomes time to generate a  */
                                    /* BFA message, we need to find the    */
                                    /* information to enter in that        */
                                    /* message. This field permits the     */
                                    /* entering of constructs of the form  */
                                    /* "$.name.name.name," where all the   */
                                    /* names except the last are           */
                                    /* references. Thus beginning with the */
                                    /* fact (DKR) that caused the trigger  */
                                    /* to fire, we may pinpoint specific   */
                                    /* fields in specific facts.           */
    char *f_dkr_default;            /* When all other sources fail, put    */
                                    /* this value in the outgoing BFA      */
                                    /* message. ASCII in the template file.*/
    struct NEXT_FRAME *num_options; /* Where we go from here; linked       */
                                    /* list of options.                    */
};
```

To fill in these data structures and properly interconnect them, the translation program follows the following procedures. The protocol name as entered via the "-p" command line argument is divided into a path and a file name. Then the protocol file is opened, the first line read, and the information stored in the PROTOCOL structure. For each message line in the protocol file, the path that was determined above is prefixed to the message file name and that file opened. A MESSAGE_HEAD structure is allocated followed immediately by the allocation of a REF_DKRS structure

```
struct REF_DKRS {
    char *r_dkr;
    char *r_dkr_fields[MAX_REF_FLDS];
    struct REF_DKRS *r_next;
};
```

After reading and storing the referenced DKRs as a linked list, processing continues with each field being processed. The following paragraphs detail this process.

As each template file is opened, a MESSAGE_HEAD structure is dynamically allocated and a pointer to this structure added to the PROTOCOL array p_loms. Then after allocating a REF_DKR structure, processing of the referenced DKRs commences. Each name is stored along with any special fields. Then the structure is added to the linked list headed MESSAGE_HEAD element ref_dkrs. If another line exists in the ref_dkrs section, the process repeats, and so on until the keyword END is reached. Note that even if there are no referenced DKRs, the END keyword must still be present.

The major task of processing the FIELD templates is next on the schedule. This process continues until the template file is exhausted, with a pointer to each dynamically allocated FIELD structure being held in a temporary array. Proper linking is delayed until all fields have been processed.

Every field template contains a first line containing the initial bit location, field width in bits, field type, and field name. This and every other line in the template file is processed by a rudimentary scanner which breaks the line into meaningful pieces. (See Appendix E-3.) After storing this data in the FIELD structure, the second line is read. This line contains the places to store incoming data and places to retrieve data for outgoing messages. The default value for the field and the name of any needed translation function are also on line two. NONE is an acceptable replacement for any of these entries. For the NUM and CHAR fields, this represents the end of translation data, and the NEXT_FRAME section

immediately follows. For the ENUM and TBL fields, the tables must still be read and stored. For ENUM fields, an array of pointers, f_etable, is allocated. This table has 2^field_width entries that in most cases are too many. (At some point, once the table has been completely processed, we should go back and reallocate this table in a more appropriate sized array.) Then as the enumerated values are read, one per line, they are dynamically stored with a pointer placed in the array, f_etable. This processing terminates when the keyword END is encountered. Processing of the TBL field table is just like the ENUM table except the the array allocated is composed of TABLE structures.

```
struct TABLE {

    int  t_index;              /* A character or integer index.    */
    char *t_value;             /* A string value.                  */
};
```

As each line is processed, the index is examined, and if the first character is a single quote, ', then the index is treated as a character index and is stored as is; otherwise the index is converted into an integer and stored as such. The table value is stored in dynamically allocated space. Again, processing continues until the keyword END is encountered.

The final field to be considered is the repeat field (RPT_I). Some BOM protocols allow for the repetition of an individual field or set of fields. Generally another field is used to specify how many times the indicated fields are to be repeated. These specifier fields are named RPT_I fields. Although the first line is the same as the data fields described above, the second line contains the number of fields to be repeated, the translation function, and the name of the field at which the repetition is to start.

The last part of each field template is the next-field section. This section ties the fields of a message together, allowing for variations in messages contents. Each line in the next-field section is stored in its own NEXT_FRAME structure.

```
struct NEXT_FRAME {

    char *ref_field;            /* Field that determines to where we   */
                                /* branch. (Field Name, SELF,          */
                                /* or ABSOLUTE).                       */
    char *ref_value;            /* Value of reference field that       */
                                /* results in this branch.             */
    char *nxt_field;            /* Name of field to which we branch.   */
    struct FIELD *next_field;   /* Pointer to field that we go to next.*/
                                /* This is filled at the end of        */
                                /* this message processing.            */
    struct NEXT_FRAME *next_frame;  /* Pointer to next element.        */

};
```

In those cases where the ref_field value is the keyword ABSOLUTE, the ref_val entry is omitted. Only names of fields are stored in the NEXT_FRAME structure as the message template is processed. After the last template field is processed, a second pass is made through the field list and actual structure pointers are inserted in the NEXT_FRAME structures. Until this time, the FIELD pointers have been held in an array. This greatly expedites searching for succeeding FIELD structures.

3.3 Address File Processing. After the address file is opened, an ADDRESS structure is allocated and anchored at the global pointer addr_data. The ADDRESS structure is shown below.

```
struct ADDRESS {

    char *a_handle;             /* Trigger Handle                      */
    char *a_dkr_type;           /* Fact type on which we are triggering.*/
    char *a_expression;         /* Expression for use in the trigger   */
                                /* copied verbatim into dfb trigger    */
                                /* command.                            */
    char *a_what;               /* What kind of message are we         */
                                /* sending when this trigger fires     */
    char *a_unit_id;            /* Unit to which we are going to send this */
                                /* message                             */
    char *a_net_addr;           /* Net address, depends on BFA         */
                                /* protocol being used.                */
    char *a_fact_ref;           /* Fact id corresponding to this unit  */
                                /* stored as a string for convenience. */
    struct ADDRESS *a_next;     /* Pointer to next one in              */
                                /* linked list.                        */
};
```

Processing the address file also requires a number of requests to the DFB, primarily to gather unit information. The query is based on the DKR field "idnum" and has the form

query unit_type ($.idnum == NNNNNN);

where NNNNNN represents the identification number contained in a_net_id. The returned DKR identification is stored in a_fact_ref. Next, a DFB "trigger" command is constructed using information supplied by the address file entry and sent to the DFB. These trigger statements look like the following:

trigger a_handle a_dkr_type (a_expression);

where a_handle, a_dkr_type, and a_expression are copied directly from the ADDRESS structure. The new ADDRESS structure is then added to the end of the linked list structure.

3.4 Message Processing. Flow of BFA BOMs in the translation program proceeds in two directions, and the processing needed, except for the sharing of the template files and translation functions, is disjoint. First the conversion of BOM to DFB commands is explained, and then the reverse will be considered.

BOM messages may arrive from either the message input file or the TTY interface. They are stored in the IN_MSG structure, p_a_message, and immediately passed to the translator function.

```
struct IN_MSG {
    int byte_cnt;                   /* Byte count of current message      */
    unsigned char in_msg[2048];     /* Place to store the current message */
};
```

In the translator function, a static MSG_HDR structure is initialized and the template for the message header is found.

```
struct MSG_HDR {

    struct IN_MSG *mh_orig_msg;     /* The original message.              */
    struct REF_DKRS *mh_header;     /* DKRs referenced in                 */
                                    /* the header.                        */
```

```
    struct REF_DKRS *mh_body;       /* DKRs referenced in              */
                                    /* the message.                    */
    struct list fields;             /* Linked list of MSG_FD           */
                                    /* structs.                        */
};
```

The fields are stored in MSG_FD structures as they are converted into DKR representations.

```
struct MSG_FD {

    char *msg_f_val;                /* Value from table NUM,ENUM.      */
    int  msg_b_val;                 /* Bit value from incoming msg,    */
                                    /* also value of NUM.              */
    int msg_e_use;                  /* Extended field value is used    */
                                    /* 1 = int, 2 = char pointer.      */
    char *msg_e_val;                /* An extra place to store         */
                                    /* values that are multifield,     */
                                    /* i.e., dtgs, unit addresses.     */
    struct FIELD *bom_ptr;          /* Pointer to appropriate          */
                                    /* template entry for this field.  */
};
```

The MSG_FDs created by this process are tied together by a linked list structure that is doubly linked. One link is in the forward direction while the second returns up the list. The details of the generic linked list and associated routines are shown in Appendix E-1. A schematic diagram of the translated field list is shown in Figure 4.

The actual translation process consists of two passes through a translation loop. The first pass processes the header, and the second completes the process with the message body. A MSG_HDR variable, message, is used to anchor the translated field list.

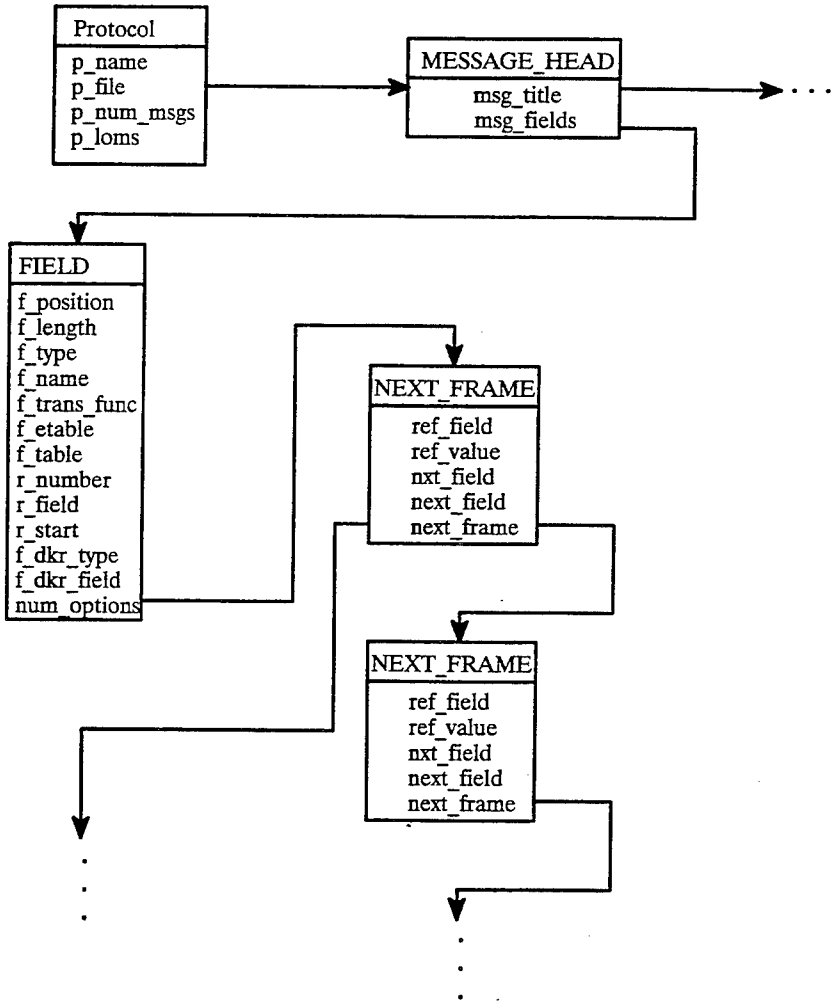


Figure 4. Translated message data structure.

For each field template, a MSG_FD structure is dynamically allocated and the indicated bits are read from the BOM message. This value read is stored directly in the msg_b_val of the MSG_FD structure and processed according to field type. For all of the various fields, if a translation function has been named, it is called with the MSG_FD and the direction flag set to FORWARD. When such a function is called, it is assumed that the called function will perform all required processing.

Processing of ENUM and TBL fields is very similar. The value stored in msg_b_val is used as an index into the table of DKR values stored with the field template. Once a match is obtained, the DKR value is copied into dynamically allocated space and the pointer stored as the msg_f_val.

CHAR and NUM fields are also processed in a similar manner. The numerical value as read from the BOM is stored in the msg_b_val. Then in the CHAR fields, the value is masked to bits and stored

as a string in the msg_f_val. For the NUM field, the msg_b_val is converted into its ACSII representation and stored in a dynamically allocated msg_f_val.

RPT_I fields are stored in the r_grp array as well as the MSG_HDR list.

```
struct REP_GRP    {

    int num_reps;                   /* Number of times to repeat        */
                                    /* this group of fields             */
    int num_fields;                 /* Number of fields in this         */
                                    /* repeat group                     */
    struct FIELD *rep_start;        /* Pointer to field template        */
                                    /* that is at the beginning         */
                                    /* of this repeat group             */
};
```

The msg_b_val is copied into the num_reps element, and the other structure elements are filled from the template for this field. The msg_f_val for this field is the ACSII representation of the msg_b_val.

Then, as each new field falls under consideration, the r_grp array is searched for a match. If a match is found, a loop is set up and the same group of template fields executed for the specified number of repetitions. SPARE fields are simply ignored.

Finally, the MSG_FD field is stored on the linked list and the NEXT_FRAME list examined to determine the next field to be processed. This process continues until the header template is exhausted. The template for the message body is then opened and the entire process repeated. The MSG_FD structures are attached to the same linked list as the header fields.

When all field templates for the header and message body have been examined, a pointer to the MSG_HDR structure is returned.

4. DKR COMMAND CREATION

After an incoming BFA message has been parsed into a list of field values under the control of the appropriate template files, the translation into DFB commands must be completed.

The first action is to examine the REF_DKRS structures in the MSG_HDR structure and clear enough elements in the static dcb array to hold the referenced DKRs. The dcb array is composed of DFB_CMD_BUF structures.

```
struct DFB_CMD_BUF {

    char *dkr_types;          /* The name of the DKR we are effecting   */
    char **add_fields;        /* Any special fields from the template   */
                              /* file                                   */
    char *fid;                /* The fact id shall be created or        */
                              /* updated                                */
    char *work_buf;           /* The buffer in which actual DFB         */
                              /* command will be constructed            */
};
```

Then the dcb array is initialized by copying the DKR names and additional fields from the REF_DKRS structures. If the r_dkr field in the REF_DKRS structure begins with an '@', it is assumed that what we have is a DKR identifier, and it is then copied into the fid field of the DFB_CMD_BUF structure. If, on the other hand, there is only a DKR name, an empty state command is sent to the DFB. This has the effect of reserving a DKR identifier for future use. This DKR_id is stored in the fid element. The final possibility is that the r_dkr field begins with a '$' character. In this case, the reference is to a field in one of the DKRs yet to be built. In the interest of efficiency, processing is delayed until all the DFB commands have been constructed.

We now start building the DFB commands. Each command will be of the form "update DKR_ID." If the fid element of the DFB_CMD_BUF structure has a value, it is inserted in the place of DKR_ID; otherwise space for the DKR_ID is reserved in the command buffer to be filled in later. Additional fields from the add_fields are also added to the end of the command to be built.

Now the processing of message field data begins. The work is done within a double loop, the outside loop being on message fields and the inside loop being across the active DFB_CMD_BUFs. Hence all DFB commands are built simultaneously. If the message field is an extended field, i.e., msg_e_use is not equal to 0, the msg_e_use variable is examined, and if the value is 1, the msg_e_val is converted into a string and stored as "fieldname = value" in the command buffer. If msg_e_use is 2, the msg_e_val value is examined, and if the field begins with the character '@' or is composed entirely of digits, the field is stored directly into the command buffers following its name. Otherwise, the value is inclosed within

double quotes and stored as above. For those cases where the msg_e_use value is 0, the values are taken from the msg_f_val element. The msg_f_val value is examined, and if the field begins with the character '@' or is composed entirely of digits, the field is stored directly in the command buffers following its name. Otherwise, the value is enclosed within double quotes and stored as above.

After all fields have been processed, each DFB command is properly terminated. Then DKR_IDs that were not known earlier are found within the constructed DKR updates and placed in the space reserved earlier. Finally the completed commands are sent to the DFB in the same order they appeared in the template file. This is particularly important for those cases where one DKR depends on information in another.

5. BFA MESSAGE CREATION

The creation of a BFA message always begins with the arrival of a notification that a trigger has fired. This notification contains the handle of the trigger that fired and the DKR identification of the DKR whose alteration caused the trigger to fire. The first action taken is to retrieve the identified DKR from the DFB. The second step is to examine the address file information and find the matching trigger handle, the type of message to be created, and to whom it is to be sent. The template structure is then searched and pointers to both the header and message body located. Message generation can now begin. A temporary buffer is zeroed, and a loop on template header fields is started. The current field template is examined and, according to the FIELD type, processed.

If the field is of type NUM, we first examine the template to see if a translation function has been named. If so, that function is called and all necessary processing is assumed to be performed within that function. The direction argument to the function is set to REVERSE and the data field argument is empty. Since a great deal of information is required to accomplish the filling of these message fields, an INFO_TRANS structure is defined to serve as an interface to the translation functions. That structure is shown below.

```
struct INFO_TRANS {

    struct ADDRESS *i_addr_info;    /* Address information for the    */
                                    /* recipient of this message.     */
    struct ADDRESS *i_addr_me;      /* The translation node           */
```

```
                                    /* address information.                */
    char *i_dkr;                    /* Triggering DKR as returned          */
                                    /* by the DFB.                         */
    struct MSG_FD  *i_history;      /* Array of field values used          */
                                    /* in determining the next             */
                                    /* field to be processed.              */
    int *i_num_fields;              /* Number of entries in                */
                                    /* history array.                      */
    struct FIELD   *i_field;        /* Template pointer for this           */
                                    /* field.                              */
    unsigned char *i_mbuf;          /* The BOM buffer.                     */
    int *i_lbit;                    /* Current location in buffer.         */
    int i_offset;                   /* Set to header size when             */
                                    /* processing message body.            */
};
```

If no translation function is named, the specified field is extracted from the DKR. This field need not be in the furnished DKR but may be indirectly referenced through that DKR. If no DKR field is specified or the specified field cannot be found, the default value is used. Two options present themselves at this point. The default value may be a number. In that case, the number is inserted into the message as is. If the default value begins with the character 'a', the field is to be taken from one of the unit_type DKRs found in the address information. In this case, the specified DKR is retrieved and the named field extracted.

The resulting value is stored in the history array and then written into the message at the appropriate place. Then the next field to be inserted into the message is determined, and processing of that field begins.

Character field processing proceeds just as for the NUM field except that no referencing through the address information is allowed.

No processing is done for SPARE fields, and nothing is written into the message array. This may result in problems if the protocol does not like fields with all the bits set to 0 (e.g., TACFIRE).

The TBL and ENUM fields are processed like the NUM and CHAR fields in that the DKR value is obtained from either a specified DKR field or the default field in the template structure. This value is then compared to the DKR values found in the template tables, and either a real index (in the case of TBL

fields) or an implied index (for ENUM fields) is found. This index is then written into the message buffer.

RPT_I fields are filled either from a referenced DKR field or a translation function. No default values are allowed.

After all the fields have been filled in for the header, the offset corresponding to the header length is set and the above process redone for the message body. After the body has been processed, the function "f_word_cnt" is called. The purpose of this function is to fill special fields such as checksums, word counts, etc. The name of this function must have been stored in the variable f_word_cnt in a translation function at some point.

Finally the message is logged if requested and passed to the protocol communications function with the OUT argument set.

5.1 DFB Interface. Communications with the DFB occur via the Package Protocol. (See Appendix E for more detail about this software.) This suite of software is based on the Berkley implementation of TCP/IP streams. Originally written by ARL personnel to allow multiple stream communications between the ARL and the Space Telescope Science Institute offices over a single physical link, the pkg protocol has proven itself ideal for connecting a DFB to its application programs. The use of transport control protocol/internet protocol (TCP/IP) allows a clean separation between the applications and the DFB. In fact, the DFB may be on a completely separate computer from the application. The only requirement made on this link is that it be high speed. Reliability is provided by the TCP/IP streams.

The application program, in this case the translation program, is provided the necessary functions to use the PKG protocol in a DFB library "libdkb". This library, the "select" system call, and the function "sp_dkb_cmd" provide all communications with the DFB. The system call "select" is used to detect input on all of the input channels available to the translation program and provides this service to the PKG routines with no additional effort. The "sp_dkb_cmd" handles the communication details of sending a command to the DFB and accepting the response. In addition, it calls a routine that performs a degree of preprocessing on the response, stripping off sequence numbers, etc., and in the case of trigger notifications, it goes ahead and retrieves the DKR that caused the trigger to fire.

5.2 TTY Interface. The easiest way to attach tactical hardware to the translation program is via an RS232 port. For the demonstration program created as part of this project, a Digital Message Device (DMD) (TM) was used to display and enter TACFIRE messages. To accomplish this interconnection, a Tactical Communication Modem (TCM) was used (Magnavox 1987). This device understands the TACFIRE protocol and serves as a smart modem allowing commercial digital computers to interface with TACFIRE hardware. The hardware configuration is shown in Figure 5 with the TCM connected to the computer via an RS232 cable and the connection between the modem and the DMD being wire line. The TCM performs the required conversion into FSK along with the time-dispersed Hamming encoding for error correction. It was also used to generate ACKs to the DMD.

To process incoming and outgoing messages into a form understandable by the TCM, a subroutine was written for inclusion in the TACFIRE library. This routine's primary job is to extract the TACFIRE message from the protocol and discard any incidental noise characters that may have been received. On the outbound side, it makes sure that required header and trailer characters are present in appropriate numbers.

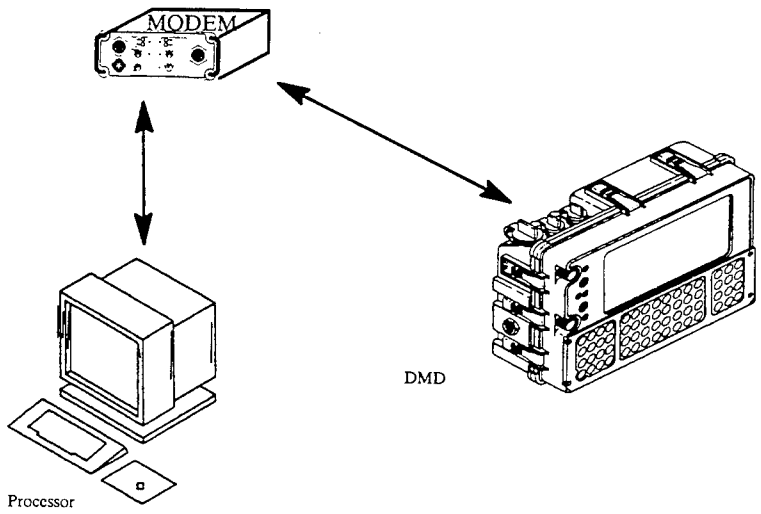


Figure 5. Hardware configuration.

6. CONCLUSIONS AND FUTURE DIRECTIONS

The results achieved in this project demonstrate that it is possible to create a translator program of generic design to perform the conversion from BOMs into a DKR. The working prototype described in this report was more than able to keep up with the tactical communication channels when converting CVC2 Call-For-Fire messages into TACFIRE FRGRID messages.

The prototypes created are based on characteristics found in the CVC2 and TACFIRE protocols and may require changing as other protocols are scrutinized. In fact, the existing prototypes should be regarded as just that, prototypes, and rewritten in the near future. The concept of using template files, while justified in this effort, did not provide as clean a description of the protocols as had been originally desired. The library routines needed to support the template files for these two protocols were extensive, particularly for the TACFIRE protocol. The primary reason for this is that TACFIRE is not really a bit-oriented protocol, but is an abbreviated character protocol. Fields are really no more than just enumerated types or integer numbers.

INTENTIONALLY LEFT BLANK.

7. REFERENCES

Chamberlain, S., M. Muuss, and J. Pistritto. "The BRL Fire Support Application for ADDCOMPE." Special Publication BRL-SP-53, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, April 1986.

Chiu, D., and E. Tuttle. "Smart Weapons Systems/Commander's Intelligent Display Final Report." December 1989.

"DDN Protocol Handbook (NIC 60004)," vol. 1. DOD Military Standard Protocol, DDN Network Information Center, SRI International, Menlo Park, CA.

"Digital Message Device (ANPSG-2A) Technical Manual." TM 11-7440-281-12&P, Department of the Army, May 1982.

General Dynamics Land Systems Division. "Combat Vehicle Command and Control Program Interface Control Document." ICD-SY10001A, Warren, MI, February 1991.

Gouchnour, D. "The Information Processor: IP - Real Time Prediction and Tracking."

Hartwig, G. W., and T. A. DiGiacinto. "The Information Distribution System - Overview of DFB and SCM Technology." (Unpublished.)

"Information Processing Systems - Open Systems Interconnection - Estelle: A Formal Description Technique Based on an Extended State Transition Model." International Standard 9074, 1989.

Johnson, S. C. "YACC - Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report no. 32, 1978.

Kaste, R. C. "Fire Advisor - An Experimental Artillery Decision Aid: Concepts and Implementation." BRL-MR-3841, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, June 1990.

Kaste, V. "The Information Distribution System. "The Fact Exchange Protocol - A Tactical Communications Protocol." BRL-MR-3856, U.S. Army Balistic Research Laboratory, Aberdeen Proving Ground, MD, August 1990.

Smith, K., and E. Heilman. "Information Distribution System Applied to Army Aviation and Air Defense Counter-Air Operations." BRL-TR-3397, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, September 1992.

Magnavox. "Operation Manual for Tactical Communications Modems." 815299-80X, rev. D, Magnavox Electric Systems Company, 9 February 1987.

Project Manager Office, Field Artillery Tactical Data Systems (FATDS). "Prime Item Development Specification for the Fire Support Team Digital Message Device (FIST DMD)." AN/PSG-5, CR-CE-0091-001A, Ft. Monmouth, NJ, 15 October 1983.

Rogers, H., and A. Ellis. "Smart Weapons System (SWS) LABCOM Cooperative Program Summary Report." BRL-MR-3926, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, July 1991.

Spencer, T. W. "undump.c - A program to Convert a Core File to an a.out." Computer Science Department, University of Utah, Salt Lake City, UT, February 1982.

Zubal, O., and M. A. Thomas. "HEL Smart Howitzer Automated Management System (SHAMS)." Technical Note 8–90, October 1990.

APPENDIX A:

TEMPLATE EXAMPLES

INTENTIONALLY LEFT BLANK.

APPENDIX A-1:

EXAMPLE TACFIRE TEMPLATES

INTENTIONALLY LEFT BLANK.

This appendix contains the example template files for the TACFIRE fixed format header and FRGRID messages. They have been used with the prototype translation program to demonstrate concept feasibility.

The top of the TACFIRE template files is tacfire.protocol.

```
tacfire.protocol 4 tf_comm
# filename      message type
#
tacfire_header  "TACFIRE HEADER"
tacfire_frgrid  "TACFIRE FR GRID"
tacfire_atigrid  "TACFIRE ATI GRID"
tacfire_freetext    "TACFIRE FREETEXT"
tacfire_acknak  "TACFIRE ACKNAK"
```

This file is the template for processing the TACFIRE message header.

```
TACFIRE HEADER
cur_msg
END
0  8  CHAR  "DESTINATION"
cur_msg msg_dst             NONE  NONE tf_addr
ABSOLUTE "NUMBER"
END
8  8  CHAR  "NUMBER"
cur_msg msg_trynum NONE       O NONE
ABSOLUTE "AUTHORIZATION1"
END
16  8  CHAR  "AUTHORIZATION1"
NONE NONE NONE        O tf_mfc_s
ABSOLUTE "AUTHORIZATION2"
END
24  8  CHAR  "AUTHORIZATION2"
cur_msg msg_auth NONE O tf_mfc_e
ABSOLUTE "MSG_TYPE"
END
32  8  TBL    "MSG_TYPE"
NONE NONE   NONE DATA tf_msg_type
    '5'          "ACKNAK"
    'D'          "DATA"
    'T'          "TEST"
    END
ABSOLUTE "ORIGINATOR"
END
40  8  CHAR  "ORIGINATOR"
```

```
cur_msg msg_src              NONE  NONE tf_addr
ABSOLUTE "DATA"
END
48  8  TBL  "DATA"
cur_msg msg_type NONE   "TACFIRE FR GRID" tf_msg_desc
    '1'         "TACFIRE FR GRID"
    '2'         "f_frshift"
    'S'         "f_mto"
    'U'         "f_focmd"
    '4'         "f_subqadj"
    '7'         "f_eom_surv"
    'J'         "TACFIRE ATI GRID"
    '8'         "f_fltrace"
    'K'         "f_atipolar"
    'M'         "f_obscloc"
    'T'         "TACFIRE FREETEXT"
    'A'         "TACFIRE ACKNAK"
     END
ABSOLUTE "TEMPLATE_END"
END
```

TACFIRE ACK and NAK messages are of the same format which is described in this template file.

```
TACFIRE ACKNAK
END
# We will not worry about ACKS and NAKS for now.
0  8   TBL "ACKNAK"
NONE NONE NONE NONE NONE
    6   "ACK"
    21 "NAK"
    END
ABSOLUTE "TEMPLATE_END"
END
```

The fr-grid TACFIRE message is used for requesting fire support when the grid coordinates of the target are known.

```
TACFIRE FR GRID
sensing
target
    src=@sensing
END
# Note:  there are a number of strange field types defined
# in TACFIRE; a decision must be made to add permissible field types
# or a KLUDGE must be fabricated.
#
#
# Following fields require a translation function.
0   8   CHAR  "DIR 10 MILS"
```

```
NONE   NONE              $.src.dirO tf_mfc10_s
ABSOLUTE "DIR 10 MILS_1"
END
8   8   CHAR  "DIR 10 MILS_1"
NONE   NONE              NONE" " tf_mfc_c
ABSOLUTE "DIR 10 MILS_2"
END
16  8   CHAR  "DIR 10 MILS_2"
sensing   dir    NONE " " tf_mfc10_e
ABSOLUTE "EAST 10 M"
END
24  8   CHAR  "EAST 10 M"
NONE NONE  $.src.east NONE tf_mfc10_s
ABSOLUTE "EAST 10 M_1"
END
32  8   CHAR  "EAST 10 M_1"
NONE   NONE              NONENONE tf_mfc_c
ABSOLUTE "EAST 10 M_2"
END
40  8   CHAR  "EAST 10 M_2"
NONE   NONE              NONENONE tf_mfc_c
ABSOLUTE "EAST 10 M_3"
END
48  8   CHAR  "EAST 10 M_3"
sensing east    NONE        NONE tf_mfc10_e
ABSOLUTE "NORTH 10 M"
END
56  8   CHAR  "NORTH 10 M"
NONE   NONE              $.src.north NONE tf_mfc10_s
ABSOLUTE "NORTH 10 M_1"
END
64  8   CHAR  "NORTH 10 M_1"
NONE   NONE              NONENONE tf_mfc_c
ABSOLUTE "NORTH 10 M_2"
END
72  8   CHAR  "NORTH 10 M_2"
NONE   NONE              NONENONE tf_mfc_c
ABSOLUTE "NORTH 10 M_3"
END
80  8   CHAR  "NORTH 10 M_3"
sensing north   NONE        NONE tf_mfc10_e
ABSOLUTE "ALT 10 M"
END
88  8   CHAR "ALT 10 M"
NONE   NONE  NONE  O tf_mfc10_s
ABSOLUTE "ALT 10 M_1"
END
96  8   CHAR "ALT 10 M_1"
NONE   NONE  NONE   " " tf_mfc_c
```

```
ABSOLUTE "ALT 10 M_2"
END
104    8        CHAR "ALT 10 M_2"
NONE NONE  NONE  " " tf_mfc10_e
ABSOLUTE "GRID"
END
112    8        TBL          "GRID"
NONE   NONE                  NONE
STD NONE
    '0'          "STD"
    '1'          "E"
    END
ABSOLUTE "TYPE"
END
120    8        TBL          "TYPE"
NONE  NONE  $.src.desc PERSONNEL tf_tgt_type
    '0'          "SPECIAL"
    '1'          "PERSONNEL"
    '2'          "WEAPON"
    '3'          "MORTAR"
    '4'          "ARTY"
    '5'          "ARMOR"
    '6'          "VEHICLE"
    '7'          "RKT/MSL"
    '8'          "SUPPLY"
    '9'          "CENTER"
    'J'          "EQUIP"
    'K'          "BUILDING"
    'L'          "BRIDGE"
    'M'          "TERRAIN"
    'N'          "ASSEMBLY"
    'O'          "ADA"
    END
SELF "SPECIAL"  "STYPE_SPECIAL"
SELF "PERSONNEL"  "STYPE_PERSONNEL"
SELF "WEAPON"  "STYPE_WEAPON"
SELF "MORTAR"  "STYPE_MORTAR"
SELF "ARTY"  "STYPE_ARTY"
SELF "ARMOR"  "STYPE_ARMOR"
SELF "VEHICLE"  "STYPE_VEHICLE"
SELF "RKT/MSL"  "STYPE_RKT/MSL"
SELF "SUPPLY"  "STYPE_SUPPLY"
SELF "CENTER"  "STYPE_CENTER"
SELF "EQUIP"  "STYPE_EQUIP"
SELF "BUILDING"  "STYPE_BUILDING"
SELF "BRIDGE"  "STYPE_BRIDGE"
SELF "TERRAIN"  "STYPE_TERRAIN"
SELF "ASSEMBLY"  "STYPE_ASSEMBLY"
SELF "ADA"  "STYPE_ADA"
```

```
END
128    8        TBL          "STYPE_SPECIAL"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "ON CALL"
    '1'         "ILL1GUN"
    '2'         "ILL2GUN"
    '3'         "ILL2GUND"
    '4'         "ILL2GUNR"
    '5'         "ILL4GUN"
    'J'         "GAS NONP"
    'K'         "GAS PERS"
    'L'         "LEAFLET"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_PERSONNEL"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "INFANTRY"
    '2'         "OBSERVATION POST"
    '3'         "PATROL"
    '4'         "WORK PARTY"
    '5'         "POSITION"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_WEAPON"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "LT M GUN"
    '2'         "ANTITANK GUN"
    '3'         "HVY M GUN"
    '4'         "RECOILLESS RIFLE"
    '5'         "POSITION"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_MORTAR"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "LIGHT"
    '2'         "MEDIUM"
    '3'         "HEAVY"
    '4'         "VERY HEAVY"
    '5'         "POSITION"
    'O'         "NOT GIVEN"
```

```
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_ARTY"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "LIGHT"
    '2'         "MEDIUM"
    '3'         "HEAVY"
    '6'         "POSITION"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_ARMOR"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "LIGHT"
    '2'         "MEDIUM"
    '3'         "HEAVY"
    '4'         "APC"
    '5'         "POSITION
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_VEHICLE"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "LT WHEEL"
    '2'         "HVY WHEEL"
    '3'         "RECON"
    '4'         "BOAT"
    '5'         "AIRCRAFT"
    '6'         "HELICOPTER"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_RKT/MSL"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "APERS"
    '2'         "LT MSL"
    '3'         "MDM MSL"
    '4'         "HVY MSL"
    '5'         "ANTITANK"
    '6'         "POSITION"
    'O'         "NOT GIVEN"
```

```
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_SUPPLY"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "AMMO"
    '2'         "PTL, OIL"
    '4'         "BRG EQPT"
    '5'         "CLASS I"
    '6'         "CLASS II"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_CENTER"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "SMALL"
    '2'         "BATTALION"
    '3'         "REGIMENT"
    '4'         "DIVISION"
    '5'         "FORWARD"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_EQUIP"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "RADAR"
    '2'         "EW"
    '3'         "SEARCHLIGHT"
    '4'         "GUIDANCE"
    '5'         "LOUDSPEAKER"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_BUILDING"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "WOODEN"
    '2'         "MASNRY"
    '3'         "CONCRETE"
    '4'         "METAL"
    '5'         "SPECIAL"
    'O'         "NOT GIVEN"
    END
```

```
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_BRIDGE"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "FOOT PON"
    '2'         "VEH PON"
    '3'         "CONCRETE"
    '4'         "WOODEN"
    '5'         "STEEL"
    '6'         "SITE"
    '7'         "RAFT"
    '8'         "FERRY"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_TERRAIN"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "ROAD"
    '2'         "JUNCTION"
    '3'         "HILL"
    '4'         "DEFILE"
    '5'         "LANDING STRIP"
    '6'         "RAILROAD"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_ASSEMBLY"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "TROOPS
    '2'         "TROOPS&VEHICLES"
    '3'         "MECHANIZED TROOPS"
    '4'         "TROOPS AND ARMOR"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
128    8        TBL          "STYPE_ADA"
sensing desc    NONE         "NOT GIVEN"  tf_tgt_stype
    '0'         "UNKNOWN"
    '1'         "LIGHT"
    '2'         "MEDIUM"
    '3'         "HEAVY"
    '4'         "MISSILE"
    '5'         "POSITION"
```

```
    'O'         "NOT GIVEN"
    END
ABSOLUTE "ATTITUDE 100 MILS"
END
136    8        TBL          "ATTITUDE 100 MILS"
sensing att  $.src.att       "0/16" NONE
    '0'         "0/16"
    '1'         "1/17"
    '2'         "2/18"
    '3'         "3/19"
    '4'         "4/20"
    '5'         "5/21"
    '6'         "6/22"
    '7'         "7/23"
    '8'         "8/24"
    '9'         "9/25"
    'J'         "10/26"
    'K'         "11/27"
    'L'         "12/28"
    'M'         "13/29"
    'N'         "14/30"
    'O'         "15/31"
    END
ABSOLUTE "DOP"
END
144    8        TBL          "DOP"
NONE   NONE                 NONE"NOT GIVEN" NONE
    '0'         "PRAND"
    '1'         "PRONE"
    '2'         "PRUG"
    '3'         "PROVER"
    '4'         "DUGIN"
    '5'         "COVER"
    '6'         "NOT GIVEN"
    '7'         "-BAD-"
    '8'         "-BAD-"
    '9'         "PRAND2"
    'J'         "PRONE2"
    'K'         "PRUG2"
    'L'         "PROVER2"
    'M'         "DUGIN2"
    'N'         "COVER2"
    'O'         "NOT GIVEN"
    END
ABSOLUTE "RAD/LGTH"
END
152    8        TBL          "RAD/LGTH"
sensing len    $.src.len  "-1" tf_wdlen
    '0'         "50"
```

```
    '1'         "100"
    '2'         "150"
    '3'         "200"
    '4'         "250"
    '5'         "300"
    '6'         "400"
    '7'         "500"
    '8'         "650"
    '9'         "800"
    'J'         "1000"
    'K'         "-BAD-"
    'L'         "-BAD-"
    'M'         "-BAD-"
    'N'         "-BAD-"
    'O'         "-1"
  END
ABSOLUTE "WIDTH"
END
160   8       TBL         "WIDTH"
sensing  wd   $.src.wd  "-1" tf_wdlen
    '0'         "50"
    '1'         "100"
    '2'         "150"
    '3'         "200"
    '4'         "250"
    '5'         "300"
    '6'         "400"
    '7'         "500"
    '8'         "650"
    '9'         "800"
    'J'         "1000"
    'K'         "-BAD-"
    'L'         "-BAD-"
    'M'         "-BAD-"
    'N'         "-BAD-"
    'O'         "-1"
  END
ABSOLUTE "STRENGTH"
END
168   8       TBL         "STRENGTH"
sensing  num  $.src.num  "-1" tf_wdlen
    '0'         "-1"
    '1'         "1"
    '2'         "2"
    '3'         "3"
    '4'         "4"
    '5'         "5"
    '6'         "6"
    '7'         "7"
```

```
    '8'         "8"
    '9'         "9"
    'J'         "10"
    'K'         "25"
    'L'         "50"
    'M'         "100"
    'N'         "200"
    'O'         "500"
    END
ABSOLUTE "SHELL/FUZE"
END
#target   rds1  $.rds1        0 NONE
176    8        TBL          "SHELL/FUZE"
NONE  NONE  NONE   "NO PREFERENCE" NONE
    '0'         "NO PREFERENCE"
    '1'         "HE/Quick"
    '2'         "HE/Delay"
    '3'         "HE/Time"
    '4'         "HE/VT"
    '5'         "WP/Quick"
    '6'         "WP/Delay"
    '7'         "WP/Time"
    '8'         "HE & WP"
    '9'         "HC Smoke"
    'J'         "CLGP"
    'K'         "Illum"
    'L'         "Cont Ill"
    'M'         "Cord Ill"
    'N'         "ICM"
    'O'         "Other"
    END
ABSOLUTE "CONTROL"
END
# Meanings for tacfire frgrid control field
#   '0'         "ADJ FIRE"
#   '1'         "Reg AF"
#   '2'         "Dest AF"
#   '3'         "FFE"
#   '4'         "Repeat"
#   '5'         "AMC AF"
#   '6'         "AMC REG"
#   '7'         "AMC DEST"
#   '8'         "AMC FFE"
#   '9'         "AMC RPT"
#   'J'         "CNO FFE"
#   'K'         "TOT"
#   'L'         "DC AF"
#   'M'         "DC FFE"
#   'N'         "EOM RAT"
```

```
#   'O'         "EOM"
184    8        TBL          "CONTROL"
target  ctl  $.ctl  "ADJ FIRE"  NONE
    '0'         "0"
    '1'         "1"
    '2'         "2"
    '3'         "3"
    '4'         "4"
    '5'         "5"
    '6'         "6"
    '7'         "7"
    '8'         "8"
    '9'         "9"
    'J'         "10"
    'K'         "11"
    'L'         "12"
    'M'         "13"
    'N'         "14"
    'O'         "15"
   END
ABSOLUTE "ANGLE"
END
192    8        TBL          "ANGLE"
NONE            NONE         NONELOW NONE
    '0'         "LOW"
    '1'         "HI"
   END
ABSOLUTE "PRIORITY"
END
200    8        TBL          "PRIORITY"
NONE            NONE         NONENORMAL NONE
    'O'         "ASGN FPF"
    '1'         "URGENT"
    '2'         "NORMAL"
   END
ABSOLUTE "TGTNUM0"
END
208    8        CHAR         "TGTNUM0"
NONE            NONE         $.tgtnum  " "  tf_mfc_s
ABSOLUTE "TGTNUM1"
END
216    8        CHAR         "TGTNUM1"
NONE            NONE         NONE" "  tf_mfc_c
ABSOLUTE "TGTNUM2"
END
224    8        CHAR         "TGTNUM2"
NONE            NONE         NONE" "  tf_mfc_c
ABSOLUTE "TGTNUM3"
END
```

```
232    8       CHAR       "TGTNUM3"
NONE           NONE       NONE" " tf_mfc_c
ABSOLUTE "TGTNUM4"
END
240    8       CHAR       "TGTNUM4"
NONE           NONE       NONE" " tf_mfc_c
ABSOLUTE "TGTNUM"
END
248    8       CHAR       "TGTNUM"
target  tgtnum NONE       " " tf_mfc_e
ABSOLUTE "VOLLEYS"
END
256    8       CHAR       "VOLLEYS"
target  num1   $.num1     " " NONE
ABSOLUTE "PRI_ZN"
END
264    8       CHAR       "PRI_ZN"
NONE           NONE       NONE" " NONE
ABSOLUTE "SPACE"
END
272    8       CHAR       "SPACE"
NONE           NONE       NONE" " NONE
ABSOLUTE "OBS_NUM"
END
280    8       CHAR       "OBS_NUM"
NONE           NONE       NONE0 tf_mfc_s
ABSOLUTE "OBS_NUM_1"
END
288    8       CHAR       "OBS_NUM_1"
NONE           NONE       NONE0 tf_mfc_e
ABSOLUTE "TEMPLATE_END"
END
```

INTENTIONALLY LEFT BLANK.

APPENDIX A-2:

EXAMPLE CVC2 TEMPLATES

INTENTIONALLY LEFT BLANK.

This is the protocol file for the CVC2 templates. All of the lines beginning with a '#' do not have template files yet. Also included in this appendix are the templates for the CVC2 header and the CFF messages.

```
cvc2.protocol 41 cvc2_comm
# filename      message type
#
cvc2_header     "CVC2 HEADER"
#mopp_st_a      "MOPP Status Alert"
cvc2_air_alert  "CVC2 AIR ALERT"
#redcon_a                "REDCON Alert"
#nbc_alert      "NBC Alert"
#warn_ord       "Warning Order"
#frago          "Frago"
cvc2_cff                 "CVC2 CALL FOR FIRE"
#call_for_cas   "Call For CAS"
#cont_rpt                "Contact Report"
#eng_upd                 "engagement Update"
cvc2_spot_rpt   "CVC2 SPOT REPORT"
cvc2_sit_rpt    "CVC2 SIT REPORT"
#bridge_rpt     "bridge Report"
#mine_lay_rpt   "minefield Laying Report"
#obs_rpt                 "obstacle Report"
#rte_rpt                 "route Report"
#per_rpt                 "personnel Status"
#ammo_rpt       "ammo Status Report/Request"
#pol_rpt                 "POL Status Report/Request"
#veh_stat                "Vehicle Status"
#nbc1_ob_rpt    "NBC 1 Observers Report"
#nbc4_c_rpt     "NBC 4 Contamination Report"
#nbc5_ca_rpt    "NBC 5 Contamination Area Report"
#sbm_rpt                 "Shell Bomb Mortar Report"
#strikewarn     "Strikewarn"
#pos_upd                 "Position Update"
#wilco                   "WILCO"
#req_rpts                "Request For Reports"
#my_c_o_ol      "Own Current Operations Overlay"
#my_f_o_ol      "Own Future Operations Overlay"
#hi_c_o_ol      "Higher Current Operations Overlay"
#hi_f_o_ol      "Higher Future Operations Overlay"
#enemy_ol       "Enemy Overlay"
#upd_enemy_ol            "Enemy Overlay Update"
#obs_ol                  "Obstacle Overlay"
#upd_obs_ol     "Obstacle Overlay Update"
#fs_ol          "Fire Support Overlay"
#upd_fs_ol      "Fire Support Overlay Update"
#fire_plan               "Fire Plan"
```

```
#sec_id                    "Sector Identification"

This is the CVC2 message header template.

CVC2 HEADER
# DKRs created or modified by this message.
cur_msg
$.cur_msg.msg_src
END
#
# SPARE fields are included for completeness but are not processed.
#
0 2 ENUM "MSG_TYP"
NONE NONE NONE NONE cvc2_msg_typ
    "text"
      "overlay"
      "spare"
      "spare"
    END
SELF "text" "MSG_DESC_T"
SELF "overlay" "MSG_DESC_O"
SELF "spare"  BAD
SELF "spare"  BAD
END
#
2 6 ENUM "MSG_DESC_T"
cur_msg msg_type NONE NONE cvc2_msg_desc
    "Reserved"
    "MOPP Status Alert"
    "CVC2 AIR ALERT"
    "REDCON Alert"
    "NBC Alert"
    "Warning Order"
    "Frago"
    "CVC2 CALL FOR FIRE"
    "Call For CAS"
    "Contact Report"
    "engagement Update"
    "CVC2 SPOT REPORT"
    "CVC2 SIT REPORT"
    "bridge Report"
    "minefield Laying Report"
    "obstacle Report"
    "route Report"
    "personnel Status"
    "ammo Status Report/Request"
    "POL Status Report/Request"
    "Vehicle Status"
    "NBC 1 Observers Report"
```

```
    "NBC 4 Contamination Report"
    "NBC 5 Contamination Area Report"
    "Shell Bomb Mortar Report"
    "Strikewarn"
    "Position Update"
    "WILCO"
    "Request For Reports"
    END
ABSOLUTE "#REP_OF_PSN"
END
#
2 6 ENUM "MSG_DESC_O"
cur_msg msg_type NONE NONE cvc2_msg_desc
    "Reserved"
    "Own Current Operations Overlay"
    "Own Future Operations Overlay"
    "Higher Current Operations Overlay"
    "Higher Future Operations Overlay"
    "Enemy Overlay"
    "Enemy Overlay Update"
    "Obstacle Overlay"
    "Obstacle Overlay Update"
    "Fire Support Overlay"
    "Fire Support Overlay Update"
    "Fire Plan"
    "Sector Identification"
    END
ABSOLUTE "#REP_OF_PSN"
END
#
8 6 RPT_I "#REP_OF_PSN"
9  cvc2_rep_handler "??? UNIT"
ABSOLUTE "WILCO"
END
#
14 1 ENUM "WILCO"
cur_msg msg_wilco NONE "WILCO not required" NONE
    "WILCO not required"
    "WILCO required"
    END
ABSOLUTE "POS_REP"
END
#
15 1 ENUM "POS_REP"
NONE NONE NONE "two words" cvc2_rep_pos
    "two words"
    "three words"
    END
ABSOLUTE "MSG_WORD_CNT"
```

```
END
#
16 16 NUM "MSG_WORD_CNT"
cur_msg msg_len NONE NONE cvc2_msg_wrap
ABSOLUTE "DST UNIT"
END
#
32 4 ENUM "DST UNIT"
NONE NONE NONE NONE cvc2_unit
    "Reserved"
    "Company A"
    "Company B"
    "Company C"
    "Company D"
    "Company E"
    "Company F"
    "Company G"
    "Company H"
    "Company I"
    "Company J"
    "Company K"
    "Company L"
    "Battalion"
    "Adjacent Battalion"
    "Brigade"
    END
SELF "Reserved"  "TEMPLATE_END"
SELF "Company A" "DST ELEMENT CO"
SELF "Company B" "DST ELEMENT CO"
SELF "Company C" "DST ELEMENT CO"
SELF "Company D" "DST ELEMENT CO"
SELF "Company E" "DST ELEMENT CO"
SELF "Company F" "DST ELEMENT CO"
SELF "Company G" "DST ELEMENT CO"
SELF "Company H" "DST ELEMENT CO"
SELF "Company I" "DST ELEMENT CO"
SELF "Company J" "DST ELEMENT CO"
SELF "Company K" "DST ELEMENT CO"
SELF "Company L" "DST ELEMENT CO"
SELF "Battalion" "DST ELEMENT BN"
SELF "Adjacent Battalion" "DST ELEMENT ABN"
SELF "Brigade" "DST ELEMENT BDE"
END
#
36 2 ENUM "DST ELEMENT CO"
NONE NONE NONE NONE cvc2_element
# Company
    "Headquarters"
    "1st Platoon"
```

```
    "2nd Platoon"
    "3rd Platoon"
    END
SELF "Headquarters" "DST INDIV CO HQ"
SELF "1st Platoon" "DST INDIV CO PLT"
SELF "2nd Platoon" "DST INDIV CO PLT"
SELF "3rd Platoon" "DST INDIV CO PLT"
END
#
36 2 ENUM "DST ELEMENT BN"
NONE NONE NONE NONE cvc2_element
    "Headquarters"
    "Slice"
    "Admin/Log"
    "Spare"
    END
SELF "Headquarters" "DST INDIV HQ"
SELF "Slice""DST INDIV SLICE"
SELF "Admin/Log""DST INDIV BN A/L"
SELF "Spare" BAD
END
#
36 2 ENUM "DST ELEMENT ABN"
NONE NONE NONE NONE cvc2_element
    "Front"
    "Rear"
    "Left"
    "Right"
    END
ABSOLUTE "DST INDIV HQ"
END
#
36 2 ENUM "DST ELEMENT BDE"
NONE NONE NONE NONE cvc2_element
    "Headquarters"
    "Slice"
    "Admin/Log"
    "Spare"
    END
SELF "Headquarters" "DST INDIV HQ"
SELF "Slice" "DST INDIV SLICE"
SELF "Admin/Log" "DST INDIV BDE A/L"
SELF "Spare" BAD
END
#
38 3 ENUM "DST INDIV CO HQ"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Co 1st Sgt"
```

```
    "FIST_V"
    "Engineering Squad"
    "ADA Squad"
    "Co Cmdr"
    "Co Exec"
    END
ABSOLUTE "DST ADD ID"
END
#
38 3 ENUM "DST INDIV CO PLT"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Platoon Ldr"
    "Wingman A"
    "Wingman B"
    "Platoon Sgt"
    "Spare"
    "Spare"
    "Spare"
    END
ABSOLUTE "DST ADD ID"
END
#
38 3 ENUM "DST INDIV HQ"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "S1"
    "S2"
    "S3"
    "S4"
    "Bn Sgt Maj"
    "Bn Cmdr"
    "Bn Exec"
    END
ABSOLUTE "DST ADD ID"
END
#
38 3 ENUM "DST INDIV SLICE"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Attack Helicopter"
    "Aviation Plt"
    "Air Defense Arty"
    "Heavy Mortar"
    "Scout"
    "ENGINEER"
    "Spare"
    END
ABSOLUTE "DST ADD ID"
```

```
END
#
38 3 ENUM "DST INDIV BN A/L"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Maintenance"
    "Medic"
    "Support"
    "Spare"
    "Spare"
    "Spare"
    "Spare"
    END
ABSOLUTE "DST ADD ID"
END
#
38 3 ENUM "DST INDIV BDE A/L"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Maintenance"
    "Medical"
    "Supply Support"
    "Transportation"
    "Commo Officer"
    "Spare"
    "Spare"
    END
ABSOLUTE "DST ADD ID"
END
41 3 SPARE "spare"
ABSOLUTE  "DST ADD ID"
END
44 4 ENUM "DST ADD ID"
cur_msg msg_dst NONE US cvc2_wrap
    "US"
    "GERMAN"
    "UNITED KINGDOM"
    "FRANCE"
    "NETHERLANDS"
    "DENMARK"
    "CANADA"
    "BELGIUM"
    END
ABSOLUTE "SRC UNIT"
END
##################
48 4 ENUM "SRC UNIT"
NONE NONE NONE NONE cvc2_unit
    "Reserved"
```

```
      "Company A"
      "Company B"
      "Company C"
      "Company D"
      "Company E"
      "Company F"
      "Company G"
      "Company H"
      "Company I"
      "Company J"
      "Company K"
      "Company L"
      "Battalion"
      "Adjacent Battalion"
      "Brigade"
      END
 SELF "Reserved"  "TEMPLATE_END".
 SELF "Company A" "SRC ELEMENT CO"
 SELF "Company B" "SRC ELEMENT CO"
 SELF "Company C" "SRC ELEMENT CO"
 SELF "Company D" "SRC ELEMENT CO"
 SELF "Company E" "SRC ELEMENT CO"
 SELF "Company F" "SRC ELEMENT CO"
 SELF "Company G" "SRC ELEMENT CO"
 SELF "Company H" "SRC ELEMENT CO"
 SELF "Company I" "SRC ELEMENT CO"
 SELF "Company J" "SRC ELEMENT CO"
 SELF "Company K" "SRC ELEMENT CO"
 SELF "Company L" "SRC ELEMENT CO"
 SELF "Battalion" "SRC ELEMENT BN"
 SELF "Adjacent Battalion" "SRC ELEMENT ABN"
 SELF "Brigade" "SRC ELEMENT BDE"
 END
 #
 52 2 ENUM "SRC ELEMENT CO"
 NONE NONE NONE NONE cvc2_element
 # Company
      "Headquarters"
      "1st Platoon"
      "2nd Platoon"
      "3rd Platoon"
      END
 SELF "Headquarters" "SRC INDIV CO HQ"
 SELF "1st Platoon" "SRC INDIV CO PLT"
 SELF "2nd Platoon" "SRC INDIV CO PLT"
 SELF "3rd Platoon" "SRC INDIV CO PLT"
 END
 #
 52 2 ENUM "SRC ELEMENT BN"
```

```
NONE NONE NONE NONE cvc2_element
    "Headquarters"
    "Slice"
    "Admin/Log"
    "Spare"
    END
SELF "Headquarters" "SRC INDIV HQ"
SELF "Slice""SRC INDIV SLICE"
SELF "Admin/Log""SRC INDIV BN A/L"
SELF "Spare" BAD
END
#
52 2 ENUM "SRC ELEMENT ABN"
NONE NONE NONE NONE cvc2_element
    "Front"
    "Rear"
    "Left"
    "Right"
    END
ABSOLUTE "SRC INDIV HQ"
END
#
52 2 ENUM "SRC ELEMENT BDE"
NONE NONE NONE NONE cvc2_element
    "Headquarters"
    "Slice"
    "Admin/Log"
    "Spare"
    END
SELF "Headquarters" "SRC INDIV HQ"
SELF "Slice" "SRC INDIV SLICE"
SELF "Admin/Log" "SRC INDIV BDE A/L"
SELF "Spare" BAD
END
#
54 3 ENUM "SRC INDIV CO HQ"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Co 1st Sgt"
    "FIST_V"
    "Engineering Squad"
    "Aid Station"
    "ADA Squad"
    "Co Cmdr"
    "Co Exec"
    END
ABSOLUTE "SRC ADD ID"
END
#
```

```
54 3 ENUM "SRC INDIV CO PLT"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Platoon Ldr"
    "Wingman A"
    "Wingman B"
    "Platoon Sgt"
    "Spare"
    "Spare"
    "Spare"
    END
ABSOLUTE "SRC ADD ID"
END
#
54 3 ENUM "SRC INDIV HQ"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "S1"
    "S2"
    "S3"
    "S4"
    "Bn Sgt Maj"
    "Bn Cmdr"
    "Bn Exec"
    END
ABSOLUTE "SRC ADD ID"
END
#
54 3 ENUM "SRC INDIV SLICE"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Attack Helicopter"
    "Aviation Plt"
    "Air Defense Arty"
    "Heavy Mortar"
    "Scout"
    "ENGINEER"
    "Spare"
    END
ABSOLUTE "SRC ADD ID"
END
#
54 3 ENUM "SRC INDIV BN A/L"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Maintenance"
    "Medic"
    "Support"
    "Spare"
```

```
    "Spare"
    "Spare"
    "Spare"
    END
ABSOLUTE "SRC ADD ID"
END
#
54 3 ENUM "SRC INDIV BDE A/L"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Maintenance"
    "Medical"
    "Supply Support"
    "Transportation"
    "commo Officer"
    "Spare"
    "Spare"
    END
ABSOLUTE "SRC ADD ID"
END
#
57 3 SPARE "spare"
ABSOLUTE "SRC ADD ID"
END
#
60 4 ENUM "SRC ADD ID"
cur_msg msg_src NONE  US cvc2_wrap
    "US"
    "GERMAN"
    "UNITED KINGDOM"
    "FRANCE"
    "NETHERLANDS"
    "DENMARK"
    "CANADA"
    "BELGIUM"
    END
ABSOLUTE "SND_GZN"
END
#
64 2 SPARE "spare"
ABSOLUTE "SND_GZN"
END
# valid 1-60
66 6 NUM "SND_GZN"
NONE NONE NONE 56 NONE
ABSOLUTE "SND_GZL"
END
#
# valid A-Z less I and O
```

```
72 8 CHAR "SND_GZL"
NONE NONE NONE E NONE
ABSOLUTE "100KM SQ_COL"
END
#
# valid A-Z less I and O
80 8 CHAR "100KM SQ_COL"
NONE NONE  NONE N NONE
ABSOLUTE "100KM SQ_ROW"
END
#
# valid A-V less I and O
88 8 CHAR "100KM SQ_ROW"
NONE NONE  NONE B NONE
ABSOLUTE "EASTING"
END
#
# 4 digit unsigned integer
96 16 NUM "EASTING"
$.cur_msg.msg_src east NONE $.addr.me.east cvc2_loc_east
ABSOLUTE "NORTHING"
END
#
# with leading zs
# addr ==> look in address file data
# me ==> the trigger id to look at
# north is the field in the referenced fact (me)
112 16 NUM "NORTHING"
$.cur_msg.msg_src north NONE $.addr.me.north cvc2_loc_east
ABSOLUTE "DTG DATE"
END
#
# valid 1-31
128 5 NUM "DTG DATE"
NONE NONE NONE CUR_TIME cvc2_dtg_date
ABSOLUTE "DTG HOUR"
END
#
# valid 0-23
133 5 NUM "DTG HOUR"
NONE NONE NONE NONE cvc2_dtg_hour
ABSOLUTE "DTG MINUTE"
END
#
# valid 1-60
138 6 NUM "DTG MINUTE"
NONE NONE NONE NONE cvc2_dtg_min
ABSOLUTE "DTG SECOND"
END
```

```
#
# valid 1-60
144 6 NUM "DTG SECOND"
NONE NONE NONE NONE cvc2_dtg_sec
ABSOLUTE "DTG MONTH"
END
#
150 5 ENUM "DTG MONTH"
NONE NONE NONE NONE cvc2_dtg_mon
    "SPARE"
    "JANUARY"
    "FEBRUARY"
    "MARCH"
    "APRIL"
    "MAY"
    "JUNE"
    "JULY"
    "AUGUST"
    "SEPTEMBER"
    "OCTOBER"
    "NOVEMBER"
    "DECEMBER"
    "SPARE"
    END
ABSOLUTE "DTG TIME ZONE"
END
#
155 5 ENUM "DTG TIME ZONE"
cur_msg msg_time NONE GOLF cvc2_dtg_wrap
    "SPARE"
    "ALPHA"
    "BRAVO"
    "CHARLIE"
    "DELTA"
    "ECHO"
    "FOXTROT"
    "GOLF"
    "HOTEL"
    "JULIET"
    "KILO"
    "LIMA"
    "MIKE"
    "NOVEMBER"
    "PAPA"
    "QUEBEC"
    "ROMEO"
    "SIERRA"
    "TANGO"
    "UNIFORM"
```

```
    "VICTOR"
    "WHISKEY"
    "X-RAY"
    "YANKEE"
    "ZULU"
    "LOCAL"
    "SPARE"
    END
"#REP_OF_PSN" 0 TEMPLATE_END
"#REP_OF_PSN" ANY "??? UNIT"
END
####################
160 4 ENUM "??? UNIT"
NONE NONE NONE NONE cvc2_unit
    "Reserved"
    "Company A"
    "Company B"
    "Company C"
    "Company D"
    "Company E"
    "Company F"
    "Company G"
    "Company H"
    "Company I"
    "Company J"
    "Company K"
    "Company L"
    "Battalion"
    "Adjacent Battalion"
    "Brigade"
    END
SELF "Company A" "??? ELEMENT CO"
SELF "Company B" "??? ELEMENT CO"
SELF "Company C" "??? ELEMENT CO"
SELF "Company D" "??? ELEMENT CO"
SELF "Company E" "??? ELEMENT CO"
SELF "Company F" "??? ELEMENT CO"
SELF "Company G" "??? ELEMENT CO"
SELF "Company H" "??? ELEMENT CO"
SELF "Company I" "??? ELEMENT CO"
SELF "Company J" "??? ELEMENT CO"
SELF "Company K" "??? ELEMENT CO"
SELF "Company L" "??? ELEMENT CO"
SELF "Battalion" "??? ELEMENT BN"
SELF "Adjacent Battalion" "??? ELEMENT ABN"
SELF "Brigade" "??? ELEMENT BDE"
END
#
164 2 ENUM "??? ELEMENT CO"
```

```
NONE NONE NONE NONE cvc2_element
# Company
    "Headquarters"
    "1st Platoon"
    "2nd Platoon"
    "3rd Platoon"
    END
SELF "Headquarters" "??? INDIV CO HQ"
SELF "1st Platoon" "??? INDIV CO PLT"
SELF "2nd Platoon" "??? INDIV CO PLT"
SELF "3rd Platoon" "??? INDIV CO PLT"
END
#
164 2 ENUM "??? ELEMENT BN"
NONE NONE NONE NONE cvc2_element
    "Headquarters"
    "Slice"
    "Admin/Log"
    "Spare"
    END
SELF "Headquarters" "??? INDIV HQ"
SELF "Slice""??? INDIV SLICE"
SELF "Admin/Log""??? INDIV BN A/L"
SELF "Spare" BAD
END
#
164 2 ENUM "??? ELEMENT ABN"
NONE NONE NONE NONE cvc2_element
    "Front"
    "Rear"
    "Left"
    "Right"
    END
ABSOLUTE "??? INDIV HQ"
END
#
164 2 ENUM "??? ELEMENT BDE"
NONE NONE NONE NONE cvc2_element
    "Headquarters"
    "Slice"
    "Admin/Log"
    "Spare"
    END
SELF "Headquarters" "??? INDIV HQ"
SELF "Slice" "??? INDIV HQ"
SELF "Admin/Log" "??? INDIV BDE A/L"
SELF "Spare" BAD
END
#
```

```
166 3 ENUM "??? INDIV CO HQ"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Co 1st Sgt"
    "FIST_V"
    "Engineering Squad"
    "ADA Squad"
    "Co Cmdr"
    "Co Exec"
    END
ABSOLUTE "??? ADD ID"
END
#
166 3 ENUM "??? INDIV CO PLT"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Platoon Ldr"
    "Wingman A"
    "Wingman B"
    "Platoon Sgt"
    "Spare"
    "Spare"
    "Spare"
    END
ABSOLUTE "??? ADD ID"
END
#
166 3 ENUM "??? INDIV HQ"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "S1"
    "S2"
    "S3"
    "S4"
    "Bn Sgt Maj"
    "Bn Cmdr"
    "Bn Exec"
    END
ABSOLUTE "??? ADD ID"
END
#
166 3 ENUM "??? INDIV SLICE"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Attack Helicopter"
    "Aviation Plt"
    "Air Defense Arty"
    "Heavy Mortar"
    "Scout"
```

```
    "Engineer"
    "Spare"
    END
ABSOLUTE "??? ADD ID"
END
#
166 3 ENUM "??? INDIV BN A/L"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Maintenance"
    "Medic"
    "Support"
    "Spare"
    "Spare"
    "Spare"
    "Spare"
    END
ABSOLUTE "??? ADD ID"
END
#
166 3 ENUM "??? INDIV BDE A/L"
NONE NONE NONE NONE cvc2_indiv
    "Reserved"
    "Maintenance"
    "Medical"
    "Supply Support"
    "Transportation"
    "Commo Officer"
    "Spare"
    "Spare"
    END
ABSOLUTE "??? ADD ID"
END
169 3 SPARE "spare"
ABSOLUTE "??? ADD ID"
END
172 4 ENUM "??? ADD ID"
cur_msg mystery_unit NONE US cvc2_wrap
    "US"
    "GERMAN"
    "UNITED KINGDOM"
    "FRANCE"
    "NETHERLANDS"
    "DENMARK"
    "CANADA"
    "BELGIUM"
    END
"POS_REP" "two words" "??? EASTING 2"
"POS_REP" "three words" "??? 100KM SQ_COL"
```

```
END
# The following is one option for representing position.  An alternative
# is two-word position with the 16 bits starting at 176 to be the easting,
# followed by the 16 bits of northing at bit position 192, and the word
# at 208 being omitted.  This choice is made according to the value of the
# field at bit position 15.
176 8 CHAR "??? 100KM SQ_COL"
NONE NONE  NONE N cvc2_loc_col
ABSOLUTE"??? 100KM SQ_ROW"
END
#
# valid A-V less I and O
184 8 CHAR "??? 100KM SQ_ROW"
NONE NONE  NONE B cvc12_loc_row
ABSOLUTE "??? EASTING 3"
END
#
# 4 digit unsigned integer
192 16 NUM "??? EASTING 3"
@"XXXXXXXX"  east NONE 9999 cvc2_loc_east
ABSOLUTE "??? NORTHING 3"
END
#
# with leading zs
208 16 NUM "??? NORTHING 3"
@"XXXXXXXX" north NONE 9999 cvc2_loc_north
ABSOLUTE TEMPLATE_END
END
#
# 4 digit unsigned integer
176 16 NUM "??? EASTING 2"
@"XXXXXXXX" east NONE 9999 cvc2_loc_east
ABSOLUTE "??? NORTHING 2"
END
#
# with leading zs
192 16 NUM "??? NORTHING 2"
@"XXXXXXXX" north NONE 9999 cvc2_loc_north
ABSOLUTE TEMPLATE_END
END
```

The call for fire template.

```
CVC2 CALL FOR FIRE
sensing
    mode = 21
target
    src = @sensing
    time = 0
```

```
END
#
# Need to set sensing mode = 21 for a call for fire message.
# Need to set target.src to the sensing FID.
#
0   1   ENUM "OF.MOC"
NONE NONE NONE "OPTIONAL FIELD NOT USED" NONE
    "OPTIONAL FIELD NOT USED"
    "OPTIONAL FIELD USED"
    END
ABSOLUTE "WARN.ORD"
END
#
1   3   ENUM "WARN.ORD"
target active $.active NONE NONE NONE
    "0"
    "1"
    "2"
    "3"
    "4"
    "5"
    "6"
    "7"
    END
ABSOLUTE "OF. T M"
END
#
4   1   ENUM "OF. T M"
NONE NONE NONE "OPTIONAL FIELD NOT USED" NONE
    "OPTIONAL FIELD NOT USED"
    "OPTIONAL FIELD USED"
    END
ABSOLUTE "TARG DESCR"
END
#
5   3   ENUM "TARG DESCR"
sensing desc $.src.desc OTHER NONE
    "OTHER"
    "BUNKER"
    "INFANTRY"
    "TANKS"
    "TRUCKS"
    "RECON"
    "SPARE"
    "SPARE"
    END
ABSOLUTE "OF. TG#"
END
#
```

```
8   1   ENUM "OF. TG#"
NONE NONE NONE "OPTIONAL FIELD NOT USED" NONE
    "OPTIONAL FIELD NOT USED"
    "OPTIONAL FIELD USED"
    END
ABSOLUTE "SIZE OF ENEMY"
END
#
9   7   NUM   "SIZE OF ENEMY"
sensing num $.src.num NONE NONE
ABSOLUTE "ENEMY EASTING"
END
#
16  16 NUM   "ENEMY EASTING"
sensing east $.src.east NONE cvc2_loc_east
ABSOLUTE "ENEMY NORTHING"
END
#
32  16 NUM   "ENEMY NORTHING"
sensing north $.src.north NONE cvc2_loc_east
"OF. TG#" "OPTIONAL FIELD USED" "TARG NUMBER"
"OF.MOC" "OPTIONAL FIELD USED" "METH OF CONT"
SELF ANY "METH OF CONT"
END
#
48  1   SPARE               "Spare"
ABSOLUTE "TARG NUMBER"
END
#
49  7   NUM   "TARG NUMBER"
target tgtnum $.tgtnum NONE NONE
ABSOLUTE "METH OF CONT"
END
#
56  2   ENUM "METH OF CONT"
target ctl $.ctl NONE NONE
        "0"
        "1"
        "2"
        "3"
        END
SELF "3" "MINUTES"
SELF "1" "MINUTES1"
"OF. T M" "OPTIONAL FIELD USED" "TYPE MUNITION"
"OF. T M" "OPTIONAL FIELD NOT USED" "TEMPLATE_END"
END
#
58  6   NUM   "MINUTES"
target time $.time NONE cvc2_cff_min
```

```
"OF. T M" "OPTIONAL FIELD USED" "TYPE MUNITION"
"OF. T M" "OPTIONAL FIELD NOT USED" "TEMPLATE_END"
END
#
58  6   NUM   "MINUTES1"
target time $.time 0 NONE
"OF. T M" "OPTIONAL FIELD USED" "TYPE MUNITION"
"OF. T M" "OPTIONAL FIELD NOT USED" "TEMPLATE_END"
END
#
64  13 SPARE                "Spare"
ABSOLUTE "TYPE MUNITION"
END
#
77  3   ENUM "TYPE MUNITION"
target rds1 $.rds1 NONE NONE
    "RESERVED"
    "DPICM"
    "HE"
    "WP"
    "SMOKE"
    "FASCAM"
    "ILLUMINATION"
    "SPARE"
    END
ABSOLUTE "TEMPLATE_END"
END
#
```

INTENTIONALLY LEFT BLANK.

APPENDIX B:

EXAMPLE LIBRARY ROUTINE

INTENTIONALLY LEFT BLANK.

The function presented here serves as a representative example of a protocol helper file for a numeric field. It demonstrates the minimum requirements in both the FORWARD (bfa->dkr) and REVERSE (dkr->bfa) directions. In the FORWARD direction, the field has already been stored in the message binary field and all we must do is required manipulation and then store the DKR value. In this particular case, we are multiplying by 10 since the incoming field value is in tens of meters and the DKR requirement is for locations in meters. In the REVERSE direction, we must first determine the value to be inserted in the outgoing message. Possibilities are examined in the following order:

1. DKR source field specified?
   a. See if that field exists.
2. If no source field, examine the default field to see if it is
   a. a message field specified.
   b. an addr field specified.
   c. a value specified.
3. If none of the above, insert zeros in the field.

```
char * cvc2_loc_east(mf, dir)
struct MSG_FD *mf;
int dir;
{

        unsigned char *cvc2_mbuf;
        struct FIELD *f;
        char *field_v;
        int itmp;
    struct ADDRESS *addr_p;
    char *tmp1;
    char *tmp2;
    char *dkr;
    char *result;

    /*
     * B F A  -->  D K R
     */

    if(dir == FORWARD){
        /* manipulate */
        itmp = mf->msg_b_val * 10;

        /* store */
        sprintf(work_buf, "%d", itmp);
      mf->msg_f_val = strdup(work_buf);
      return(NULL);
    }
```

```
/*
 *  D K R  --> B F A
 */

if(dir == REVERSE){
f = info_trans.i_field;
if(f->f_dkr_source != NULL)
   field_v = field_find(info_trans.i_dkr,f->f_dkr_source);
else
   field_v = ERROR;
if((field_v == ERROR) && (f->f_dkr_default != NULL))
   field_v = strdup(f->f_dkr_default);
if(*field_v == '$'){
   tmp1 = field_v;
   tmp1++;
   tmp1++;

   if(strncmp(field_v, "addr", strlen("addr"))){
      /* Have to look up this number from a fact referenced   */
      /* through the address information file. The construct  */
      /* is "$.addr.handle.field."  So we must pick out the   */
      /* appropriate parts from the address information and   */
      /* then query the FactBase for the appropriate field    */
      /* information.                                         */

      /* move by address file indicator "addr"                */
      while(*tmp1 != '.')
            *tmp1++;
      tmp1++;
      /* get trigger handle */
      tmp2 = tmp1;
      while(*tmp1 != '.')
            *tmp1++;
      *tmp1++ = '\0';
      /* look up and get the fact id;  tmp points to the field */
      /* of interest.                                         */
      if(STRNCMP(tmp2, "me", strlen(tmp2)) == 0)
            addr_p = info_trans.i_addr_me;
      else { /* look it up */
            addr_p = info_trans.i_addr_info;
            while(strncmp(addr_p->a_handle, tmp2, strlen(tmp2)) != 0 )
              addr_p = addr_p->a_next;
      }
      /* build the dfb command */
      sprintf(work_buf, "get %s;", addr_p->a_fact_ref);
      sp_dkb_cmd(&dkr, work_buf, 0);
      /* put field name in proper syntax */
      work_buf[0] = '$';
      work_buf[1] = '.';
```

```
            work_buf[2] = '\0';
            strcat(work_buf, tmp1);
            field_v = field_find(dkr, work_buf);
            free(dkr);
        } else
        field_v = field_find(info_trans.i_dkr,f->f_dkr_source);
}

/* At this point, we either have a value or don't.  If we do, */
/* convert it into binary and write it into the message.  If */
/* we don't, leave the field full of binary zeros.*/
if(field_v != ERROR){
        sscanf(field_v, "%d", &itmp);
        itmp /= 10;  /* convert the DKR version into terms of meters */
        free(field_v);
        sprintf(work_buf, "%d", itmp);
            field_v = strdup(work_buf);
}
st_and_wr(&info_trans, f, itmp, field_v);
}
free(field_v);
}
st_and_wr(it, f, num, ch)
struct INFO_TRANS *it;
struct FIELD *f;
unsigned int num;
char *ch;
{
    if(ch == ERROR){
       it->i_history[*(it->i_num_fields)].bom_ptr   = f;
       it->i_history[*(it->i_num_fields)].msg_b_val = 0;
       it->i_history[*(it->i_num_fields)].msg_f_val = strdup("0");
       (*it->i_num_fields)++;
       (*it->i_lbit) = f->f_position + f->f_length;
          return(NULL);
    }
    it->i_history[*(it->i_num_fields)].bom_ptr   = f;
    it->i_history[*(it->i_num_fields)].msg_b_val = num;
    it->i_history[*(it->i_num_fields)].msg_f_val = strdup(ch);
    (*it->i_num_fields)++;
    write_bits(it->i_mbuf, f->f_position+it->i_offset, f->f_length, num);
    (*it->i_lbit) = f->f_position + f->f_length;
    return(NULL);
}
```

INTENTIONALLY LEFT BLANK.

APPENDIX C:

TRANS_P MANUAL PAGE

NAME
TRANS_P - a BOM interpreter for the IDT

SYNOPSIS

trans_p [-p protocol_file] [-l] [-a address_file] [-i message_file] [-t tty] [-T time] [-h hostname] [-d] [-r] [-D] [-c] [-L log_file]

OPTIONS

-p protocol_file	File containing the protocol top-level template. It is a required argument.
-l	Load template files only. Used to verify that protocol template files are at least grossly correct.
[-a address_file]	Address file name. Contains the addresses of other units on this net and criteria for sending them messages.
[-i message_file]	A file of BOMs to be interpreted and processed as specified by other arguments.
[-t tty]	The tty port at which actual interface hardware is connected. This port will be monitored for incoming BOMs and appropriate outgoing messages will be sent over this port.
[-T time]	A time delay between processing messages found in the message_file. Used when the translation program is serving as a scenario driver.
[-h hostname]	When the 'd' flag is present, this flag is used to specify the host running the DFB. The default is the local host.
[-d]	Connect to a DFB. When a BOM is received, fact updates will be sent to this DFB.
[-r]	Read to run flag. This flag indicates that a character must be typed at the keyboard before a message from the message_file is processed. Allows finer control than with the 'T' flag.
[-D]	Turn on debugging prints for lots of reading enjoyment.
[-c]	Print on the console human readable versions of each BOM processed.
[-L log_file]	Log each BOM message received.

ENVIRONMENT

TRANS_P currently uses no variables from the environment.

DESCRIPTION

Currently the TRANS_P software executes on SUN Microsystems computers under the SUN 4.0.3 UNIX (UNIX is a trademark of AT&T) Operating System.

TEMPLATE FILE

The determining factor with the translation program is the template files. The translation program, when combined with the template files and associated library functions for a particular protocol, provides an interface between that protocol and a DFB node.

The template file organization begins with a protocol file. The protocol file contains the names of various messages within the protocol and the respective template files for interpreting them. Also, specified within the protocol file is the name of a function that is used to provide any required hardware handshaking. The protocol file used with the demonstration set of TACFIRE messages is as follows:

```
tacfire.protocol      5 tf_comm
# filename            message type
#
tacfire_header        "TACFIRE HEADER"
tacfire_frgrid        "TACFIRE FR GRID"
tacfire_atigrid       "TACFIRE ATI GRID"
tacfire_freetext      "TACFIRE FREETEXT"
tacfire_acknak        "TACFIRE ACKNAK"
```

The first element is just the name of the protocol being described and serves as a check. The second element is the number of messages, including the header template, named in this file. The third and final element is the name of a library function to handle handshaking with BFA-specific hardware. A specific instance of this function is presented elsewhere in this report. Lines beginning with '#' characters are comments. Subsequent lines relate protocol messages with their corresponding template files.

Each message template file consists of three parts. The first is simply the message name as specified in the protocol file; it is used as a check that this is indeed the proper file. The second part is a listing of DKR types affected by this message. The listed DKR types are those created when an incoming BFA message is received. Under each DKR type are fields and values that are to be added to the DKR created. These are typically control fields that do not appear in the incoming message. For instance, the DKR section for a CVC2 CALL FOR FIRE message is:

```
sensing
        mode = 21
target
        src = @sensing
        time = 0
END
```

Two DKRs will be created when a CALL FOR FIRE is received. The "sensing" DKR will have the field mode filled with the value 21; the "target" DKR will have the field time set to zero; and the field src filled with the DKR identifier of the sensing DKR; thus the target field src will be a pointer to the sensing DKR. The keyword END terminates this section.

The final and by far largest section is the actual message field descriptors. These descriptors not only describe how to interpret an incoming BFA message but how to create a BFA message as well. When this program was being created, the first protocol examined was CVC2. In this protocol, most fields contain small integers that are then used to look up the appropriate value in a table. These are better known as enumerated fields. Occasionally, numeric fields are to be interpreted literally. Some of the numeric fields are special in that they contain control information such as the number of instances of a repeated field. These original field types were called ENUM, NUM, and RPT_I. Later as the fixed format TACFIRE protocol was considered, it became clear that additional field types would be needed. Since fixed format TACFIRE is really character oriented with a limited character set, it became necessary to add the CHAR and TBL field types. Therefore, the current recognized fields are as follows:

ENUM	Number from field used as an index into a table to get actual meaning.
RPT_I	Specifies the number of times to repeat other fields in this message.

NUM	No special meaning—numeric interpretation.
CHAR	Interpreted as a 7-bit ASCII character.
TBL	Uses the message value as an index into a table to obtain the true meaning. Different from the enum type in that field values may not be sequential.
SPARE	Unused field indicator.

These fields were selected as part of an evolutionary process and contain overlaps as well as some gaps. For instance, in retrospect, the ENUM and TBL fields could be combined in a single TBL field. Also needed is a freetext type of field which allows any arbitrary entry. As a result of these oversights as well as other special cases found only after development had begun, a number of subterfuges are used to implement work arounds as required. These will be detailed later as the individual field templates are described.

The individual fields in each message are defined by entries in the template file. The form of these entries depends on the type of field being described. All fields have the same first line. This line specifies the field within the message, specifies the type of field, and names the field. The four elements on this line are as follows:

FIRST LINE - REQUIRED FOR ALL FIELDS

f_position	Bit position in the message at which this field begins. Assumes that the message is a bit stream numbered starting at 0.
f_length	Bit length of this field. Includes the initial bit position.
f_type	Type of this field, i.e., ENUM, TBL, etc.
f_name	Message name of this field.

The second line of the entry is also required, but only for data fields. This line specifies where incoming message data is to be stored, from where outgoing data is to be retrieved, and how they are to be processed. The control field RPT_I does not have this line.

SECOND LINE - REQUIRED FOR ALL DATA FIELDS

f_dkr_type	The DKR to which this data field corresponds. With the f_dkr_field, completely specifies where the information in this message field is to be stored.
f_dkr_field	The data kernel element in which to store this message field.
f_dkr_source	When it becomes time to generate a BFA message, we need to find the information to enter in that message.
f_dkr_default	Value used in the outgoing BFA message when all other sources fail.
f_dkr_func	The name of the function required to process the message field into a format compatible with the DKR.

The f_dkr_type is typically just the name of a DKR, but two other options are available. If the message field has no corresponding DKR representation, then the keyword NONE may be inserted. If this option is used, the f_dkr_field must also have the value NONE inserted. The second option was created to accommodate the fact the CVC2 protocol places the sender's location in the message header. The construct "$.cur_msg.msg_src" allows the template to reference values in the cur_msg DKR. In this case, the specific DKR representing the sending unit is specified.

The f_dkr_source field is the source of information to be placed in this field in an outgoing message. All referencing must begin from the triggering DKR. An entry of the form "$.fooy" would cause the value for the field called "fooy" to be used as the field value in the BFA message. Syntax of the form "$.name1.name2.fooy," where all the names except the last are references, would cause a path to be traced through the referenced DKRs to the field "fooy." Thus, beginning with the DKR that caused the trigger to fire, one may pinpoint specific fields in specific facts for information to insert in BFA messages. The keyword NONE may be used when the information to fill this field cannot be found as a DKR entry.

The f_dkr_default entry is, as the name implies, the value used to fill in a BFA message field when other options have been exhausted. This value should be human intelligible. It is generally the value found in the DKR. If the value is more than a single word, it should be placed within double quotes. To accommodate the sender address found in the CVC2 header, the construct "$.addr.me.east" is allowed. This causes the program to look in the address file information under the trigger name "me," then look up the appropriate DKR and use the value found in the field "east."

This last element f_trans_func is a protocol-specific library function. These functions are basically helpers, enabling more detailed data processing than is available to the general field processing software. These functions have to operate bidirectionally, i.e., forword (BFA->DKR) and reverse (DKR->BFA). They are often used where it is necessary to alter the data as interpreted or if several fields on one side coalesce into a single field on the other.

ENUM FIELD TYPES

SUBSEQUENT LINES

table The table of strings representing appropriate entries for this field. The value found in the message is used as an index into this table.

The above explanation is straightforward. The table should be terminated with the keyword END.

TBL FIELD TYPES

SUBSEQUENT LINES

table The table consisting of index values followed by the DKR value.

The table is composed of lines, each of which consists of an index followed by a value. If the index is enclosed within single quotes, then it is interpreted as an ASCII character and the numeric value of that ASCII character is inserted into the BFA message. If, on the other hand, the index is not enclosed within quotes, then the index is interpreted directly as a number and that value inserted into the outgoing message. See examples in Appendix A-1. The table should terminate with the keyword END.

RPT_I FIELD TYPES

OPTIONAL SECOND LINE

f_repeat The number of times that the specified fields are to be repeated. (This is not needed here. The number will be stored wherever the message data is stored.)

SUBSEQUENT LINES

f_repeat_names The names of the subsequent fields to be repeated.

The table should be terminated with the keyword END.

CHAR FIELD TYPES
NUM FIELD TYPES

These field types do not have any special lines.

Following the above lines, each field description has a set of lines determining which field is to be processed next. This procedure was adopted to handle the case of option fields or cases where the field size depends on the value of a previous field. Examples are found in Appendix A-2. The syntax for these control lines is reference field reference value new_field where reference field is the name of a previously processed field or one of the keywords ABSOLUTE or SELF. Lines with the ABSOLUTE keyword have no reference value since the field named in new_field is processed next absolutely with no dependence on prior fields. The keyword SELF implies that subsequent processing depends on the value of the field currently being processed. The reference value is the value that must be matched. This value must not be the index value, but rather the human readable or DKR value. The key word ANY may be used in this field. In this case, any value will cause a match. New_field is the name of the next field to be processed. The keyword TEMPLATE_END denotes the end of processing for this message. Any line entries that consist of more than one word must be enclosed in double quotes. The order of lines is important since the first match will determine subsequent processing. An unsuccessful match will cause problems and probably cause catastrophic failure. This section terminates with the keyword END.

ADDRESS FILE

The address file is used to specify communications on the BFA side of the translation program. Since all outgoing BFA messages are the result of incoming information, the DFB "trigger" mechanism is used to key BFA message creation. The "trigger" mechanism allows an application program to tell the DFB what information it is interested in. A trigger request consists of an identifier, a DKR type, and a criteria. When information enters the DFB, causing the creation or alteration of the named DKR type and meeting the specified criteria, a notification is sent to the application program. The translation program, upon receipt of the notification, retreives the triggering DKR and generates the specified BFA message.

All information required to generate the outgoing BFA is contained in the address file in association with the templates for the particular BFA under consideration. The format for the address file is as follows:

trigger_name DKR_type expression msg_type unit_id unit_address

where:

trigger_name — is an identifying name for this trigger. It is used by the DFB to identify the trigger to the application program.

DKR_type — is the DKR type to which this trigger applies. Whenever a DKR of this type is created or modified, the following criteria will be tested.

expression — is the triggering expression. Whenever this expression is satisfied, a notification is sent to the stating application program. This expression is copied directly into the trigger statement sent to the DFB.

msg_type — is the message type.

unit_id — is the idnum for a particular unit. This number is found in the DKRunit_type.

unit_address — is the unit address as determined by the BFA protocol being used.

An example address file is shown here:

```
# This is a test file for the CVC2 node.  Addresses reflect the CVC2
# addressing scheme.
#
#Trig_handle  Fact_type  Trig_expr  Msg_type  Unit_name  Unit_addr
#
#       2-11 BN TOC (OPNS)
me      NONE "($.idnum ==
#
#B/1-440 AIR DEFENSE BN
#
trig_1  sensing ($.mode==22)  "CVC2 SPOT REPORT"  B3221211  0x1300
#
```

Lines beginning with a '#' character are comments and may contain any useful information. The first data line is a special line and identifies the translation program's address on the BFA network. On this line, only the trigger handle, unit_id, and unit_address are used. No trigger is built for this line. The

expression on this line is simply to serve as an example and serves no other purpose. The unit_addr field is copied directly in the source field of the outgoing messages and therefore is dependent on the protocol. In this example, the unit_addr is binary and maps into several fields in CVC2 header. Note that the leading "0x" in the unit_addr means that this is a hexadecimal representation of the binary address.

In this example, when the DFB receives a sensing DKR update that has the mode variable set to 22, a notification will be sent to the translation program which will then generate a "CVC2 SPOT REPORT" that is then sent to the B/1-440 AIR DEFENSE BN at CVC2 address 0x1300.

FILES:

protocol_file, address_file, message_file, template_files

SEE ALSO:

dfb(1)

BUGS

Note the TRANS_P is research software and as such contains many bugs and inefficiencies.

APPENDIX D:

PACKAGE PROTOCOL SUMMARY

INTENTIONALLY LEFT BLANK.

Acknowledgment

This documentation provides a description of the libpkg routines. The original version of this writeup was kindly provided by Peter B. Shames, and has been updated to reflect BRL extensions since the first writing. It is intended that an RFC be published at some point, but for now, this report (plus the source code) represents all the documentation available. In the CAD distribution, the remote framebuffer interface (libfb/if_remote.c) and server (rfbd) communicate using this protocol. In IDT software, these routines connect the DFB with application programs.

Contributors:

BRL	U.S. Army Ballistic Research Laboratory
STScI	Space Telescope Science Institute

Pkg Interface Description

[created: 26 sept 1986: pmb shames]
[updated: 14 nov 1986: pc dykstra]
[updated: 20 dec 1991: gw hartwig]

Overview

A prototype interface that implements a fairly simple message-passing abstraction between cooperating processes running over TCP connections has been developed by Mike Muuss, Charles Kennedy, and Phillip Dykstra at the BRL. Some experiments have been run at STScI to implement this interface on a DECNET link with good success. A few changes have been suggested to improve network throughput for data flows that consist of short message traffic or byte-stream traffic.

This note describes the motivation for developing such an interface and describes the subroutine interfaces and functionality. The interface routine descriptions and portions of the textual descriptions have been taken directly from the source code provided by Muuss and Kennedy, which has been the only available description to date.

Why a message interface?

The interface PKG provides a simple interface for managing client/server connections in a distributed environment. A server task is initialized to wait for incoming requests from clients. Clients open a connection to the server task which executes on some host processor waiting for incoming requests. Messages with identifying type codes are passed between processes, with optional appended data blocks. The interface is nonblocking and accepts arbitrary data messages of arbitrary length.

For example; the interaction with a remote data archive may require request for services, user authentication, permission granting, index hierarchy queries with manipulation of results, archive request processing, and delivery of output products—electronically or otherwise. The archive is a server that responds to service requests from clients. How many may be served at one time and whether public access and private services are supported is an implementation issue. Since no direct interactive support is required, the message paradigm can support the requirements.

However, Athena's XWindow uses a client-server model to provide screen/window management services for distributed processes. XWindow is very permissive of different window management policies and provides hooks for implementing various user interfaces. At the lowest level, XWindow requires a reliable duplex byte-stream, which can be implemented in many operating environments. Because XWindow wants to manage all process-window communications directly, it builds its own block stream protocol to manage the connection. This permits the interactive update required by some processes, but basically is the same paradigm as PKG, defined within the window system.

PKG Interface Description

The PKG interface implements a client-server network connection that multiplexes synchronous and asynchronous messages across stream connections. Connection control is provided by open/close routines, where the client authentication is handled prior to establishing the connection. Server side actions include initialization, client connection, and client request servicing. Client side actions include server request, reply processing, possible connection negotiation, and data transfer processing.

A connection is established when a client issues an open request to a server, running on some host. Which hosts provide required services must be determined prior to connection requests, using some service directory. Default connection processing is stream oriented, where a process may get data as it is available on the connection. A blocking read may also be issued, waiting for any complete message, or just for

messages of a specified type. The connection remains open until explicitly closed by either client or server.

Once the connection is established, it may be used to process a full-duplex asynchronous byte stream, or a buffered, synchronous query/reply exchange. Message-type multiplexing supports user actions based upon message type, where the multiplexing is handled in the interface. Different message-type streams may operate using different flow control over the same connection. Multiplexing is handled in the interface by prepending a header block to the message before transmitting it to the reader. Data are transmitted as received, but the header is transmitted in Internet network byte/bit order.

A package connection may incur significant overhead if many short messages are passed between processes, so a buffered stream connection may be requested. The application calls the pkg_stream interface rather than the usual pkg_send request, and PKG builds a message buffer and passes it to the client as it gets filled. A stream buffer may be flushed with an explicit call to the pkg_flush routine.

PACKAGE FUNCTION DEFINITIONS

Control Interfaces -

PKG_OPEN	Open a connection to a host.
pkg_open	(host, service, protocol, uname, passwd, pkg_switch, errlog) returns ptr_to_connection. We are a client, making a connection to a server running on some host. The return value is the connection handle. Protocol, username, and password are optional.
PKG_CLOSE	Close a connection.
pkg_close	(ptr_to_connection) returns VOID. We are a client or server wishing to gracefully close a connection.
PKG_TRANSERVER	Become a one-time network server on the already accepted connection.
pkg_transerver	(pkg_switch, errlog) returns ptr_to_connection.

Given an established connection, this routine creates a pkg_conn structure for it and initializes it to use the given pkg_switch and error logging function. This routine is needed by servers which get created by some other process after the connection is established. Processes started by the UNIX "inetd" are one example.

PKG_PERMSERVER Create a network server and listen for connections.

pkg_permserver (service, protocol, backlog) returns listen_file_desc

The service port is determined, and a listen is hung on the port that is bound to the socket. The return value is the file descriptor of the socket. For this kind of server, pkg_getclient is used to accept connections from clients.

PKG_GETCLIENT Used by a server to accept a client connection.

pkg_getclient (listen_file_desc, pkg_switch, errlog, nodelay_flag) returns ptr_to_connection or NULL or ERROR

If possible, the connection request from the client is accepted, a pkg_connection block is created, and a ptr to it is returned to the server. The server may request nonblocked service by setting the nodelay_flag TRUE.

I/O Interfaces -

PKG_PROCESS Called to process all PKGs that are stored in the internal buffer pkc_inbuf.

pkg_process (ptr_to_connection)

Returns -

<0 some internal error encountered; DO NOT call select() next.
0 All ok, no packages processed
>0 All ok, return is # of packages processed (for the curious)

This routine should be called to process all PKGs that are stored in the internal buffer pkc_inbuf. This routine does no input/output (I/O), and is used in a "polling" paradigm.

Only after pkg_process() has been called on all PKG connections should the user process suspend itself in a select() operation, otherwise packages that have been read into the internal buffer will remain unprocessed, potentially forever.

If an error code is returned, then select() must NOT be called until pkg_process has been called again.

A plausible code sample might be:

```
for(;;) {
        if(pkg_process(pc) < 0) {
        printf("pkg_process error encountered\n");
        continue;
        }

        if(bsdselect(pc->pkc_fd, 99, 0) != 0) {
        if(pkg_suckin(pc) <= 0) {
        printf("pkg_suckin error or EOF\n");
        break;
        }

        }
        if(pkg_process(pc) < 0) {
        printf("pkg_process error encountered\n");
        continue;

        }
        do_other_stuff();

}
```

Note that the first call to pkg_process() handles all buffered packages before a potentially long delay in select(). The second call to pkg_process() handles any new packages obtained either directly by pkg_suckin() or as a byproduct of a handler. This doublechecking is absolutely necessary, because the use of pkg_send() or other pkg routines, either in the actual handlers or in do_other_stuff(), can cause pkg_suckin() to be called to bring in more packages.

PKG_SUCKIN — Read all data from the operating system into the internal buffer.

pkg_suckin (ptr_to_connection)

Returns -
-1 on error
0 on EOF
1 success

Suck all data from the operating system into the internal buffer. This is done with large buffers, to maximize the efficiency of the data transfer from kernel to user.

It is expected that the read() system call will return as much data as the kernel has, UP TO the size indicated. The only time the read() may be expected to block is when the kernel does not have any data at all. Thus, it is wise to call this routine only if:

a) select() has indicated the presence of data, or

b) blocking is acceptable.

This routine is the only place where data is taken off the network. All input is appended to the internal buffer for later processing.

Subscripting was used for pkc_incur/pkc_inend to avoid having to recompute pointers after a realloc().

PKG_SEND — Send a message on the connection.

pkg_send (message_type, data_buff, buff_len, ptr_to_connection)
returns bytes_sent or ERROR

Send constructs a message header for the data buffer and transmits it on the connection. If only part of the message can be sent, the actual number of bytes transmitted is returned.

Any data in the stream buffer is first flushed. If the stream buffer needs flushing, and the message is less than MAXQLEN (currently 512) bytes long, and sufficient room is left in the stream buffer, this message gets "piggybacked" by copying it to the buffer before flushing. If available, "writev" is used to send the header and data with one syscall.

PKG_2SEND — Send a two-part message on the connection.

pkg_2send — (message_type, data_buff1, buff1_len, data_buff2, buff2_len, ptr_to_connection)

Returns — bytes_sent or ERROR

2Send performs the same job as pkg_send except the data come in two parts. This is often the case for, say, a command followed by user data. User data copies can be avoided while still allowing a single trip in and out of the kernel on the receiving end.

PKG_STREAM — Send a buffered data stream on the connection.

pkg_stream — (message_type, data_buff, buff_len, ptr_to_connection)

Returns — bytes_sent or ERROR

Pkg_stream connections provide a low network overhead connection where interactive responsiveness is not required.

Connections have a PKG_STREAMLEN (currently 4096) byte-stream buffer. If the current message is less than MAXQLEN (currently 512) bytes in length, and space remains for it, it will be appended onto the end of this buffer. Otherwise the buffer is implicitly flushed by sending this message via PKG_SEND.

PKG_FLUSH — Flush the stream buffer out on the connection.

pkg_flush — (ptr_to_connection) returns bytes_sent or ERROR.

Flush takes the current contents of the stream buffer being constructed by stream and flushes it to the connection. This allows programs explicit control over the stream interface where required.

PKG_BLOCK — Wait for a complete message of any type.

pkg_block — (ptr_to_connection) returns message_arrived or ERROR.

This routine blocks, waiting for a complete message of any type to arrive on the connection. The user message handler is called to process the message. This routine can be called in a loop waiting for asynchronous messages or for the arrival of messages of uncertain type.

PKG_WAITFOR Wait for a message of specific type; return in user buffer.

pkg_waitfor (message_type, user_buff, buff_len, ptr_to_connection)
returns buff_len or ERROR.

This routine does a blocking read on the connection until a message of message_type is received. Asynchronous messages and messages of other types are processed while this routine waits. The message is returned in the user's buffer.

PKG_BWAITFOR Wait for a message of specific type; return in allocated buffer.

pkg_bwaitfor (message_type, ptr_to_connection)

Returns ptr_to_buffer or ERROR

This routine does a blocking read on the connection until a message of message_type is received. Asynchronous messages and messages of other types are processed while this routine waits. The message is returned in a newly allocated buffer that the caller must free.

User Message Handler Interface -

PKG_SWITCH Table of User Message Handler Pointers

```
struct pkg_switch {
        unsigned shortpks_type;/* Type code */
        int  (*pks_handler)();/* Message Handler */
        char*pks_title;/* Description */
};
```

pks_handler (ptr_to_connection, message_buff) returns ignored int.

The user may specify handlers for one or more message types. When these routines are called, they are passed from a pointer to the connection and from a pointer to a buffer that contains the message. The user message handler routine is responsible for freeing the message buffer after processing it. If there is no message handler for that message type, the message is discarded and the buffered freed. The switch routine uses an external message handler array, created by the caller, that defines message types, handler entry points, and a message descriptor field.

The pkg_switch list which is passed to some of the above functions is a NULL terminated array of structures containing the message type, a pointer to an function which handles messages of that type, and a descriptive string. A separate pkg_switch array is kept for each connection so that several entirely different PKG conversations can be carried out by a single application at the same time.

PKG_CONN Connection block structure

```
struct pkg_conn {

        pkg_file_desc, pkg_magic, mess_len, ptr_to_buffer,
        curr_buffer_pos, pkg_switch, errlog, stream_buff,
        curr_stream_pos
};
```

Connection block structures are created by pkg_open or pkg_getclient when a connection is established, and are freed when a connection is closed. This connection block is used during all I/O requests and contains the pointer to the start of the current buffer and to the current position in the buffer for incomplete read or writes. The mess_len field specifies the number of bytes expected by get.

The pkg_switch array, and a pointer to an error logging function for this connection are included. The stream buffer is also contained in the structure.

See pkg.h for the actual names of the structure elements.

Implementation Notes

At least two implementations of the PKG interface have been done, one based on TCP/IP connections, the other based on DECnet connections. Future releases of PKG will be layered on top of a virtual stream library known as NET.

The implementation running on 4.2BSD UNIX takes advantage of the "writev" syscall which can output the header and data in a single syscall/TCP_PUSH cycle. The advantage of the short data copy is realized only when writev can not be used, or when a pkg_flush is not required (a la PKG_STREAM).]

Future Developments

Further development of the lower levels of the interface will be provided so that this message-passing abstraction can be provided on top of both DECnet and TCP/IP network layers, with other implementations provided as required. The program interface layer is to be as described in the body of this note; the lower layers will interface to the network layers using whatever subroutine interfaces are

most appropriate for the system hosting the interface. A simple read/write/open/close abstraction may be provided at a lower layer to further isolate the network layers from the message-handling layer.

Where it is necessary to cross network protocol boundaries, message relay processes will be created to reside on a Janus host that straddles the two domains. Domain routing and relay point information will be maintained above the network layer within the extended PKG interface. Clients and servers will need to determine server host id information from some available database. Server names must either carry domain information, or such information must be available as part of the server database. Within the extended PKG interface, routing information for domain boundary transits must be maintained, and appropriate Janus host systems connected to provide relay services.

It is not anticipated that full asynchronous I/O implementation will work particularly well across domain boundaries, but all other forms of message transfer should work quite well. As already noted, buffered stream I/O may be perfectly satisfactory for many applications where full interaction is not required. Where applicable, this mode of interface is to be preferred over full asynchronous I/O. Further development of this interface layer on other protocol families is possible, perhaps even on low-level networks like BITNET and certainly on connection-oriented X.25 or OSI networks.

Functions comprising the pkg_library

pkg_gshort	Get a 16-bit short from a char[2] array.
pkg_glong	Get a 32-bit long from a char[4] array.
pkg_pshort	Put a 16-bit short into a char[2] array.
pkg_plong	Put a 32-bit long into a char[4] array.
pkg_open	Open a network connection to a host/server.
pkg_transerver	Become a transient network server.
pkg_permserver	Create a network server, and listen for connection.
pkg_getclient	As permanent network server, accept a new connection.
pkg_close	Close a network connection.
pkg_send	Send a message on the connection.
pkg_2send	Send a two-part message on the connection.
pkg_stream	Send a message that doesn't need a push.
pkg_flush	Empty the stream buffer of any queued messages.
pkg_waitfor	Wait for a specific msg, user buf, processing others.
pkg_bwaitfor	Wait for specific msg, malloc buf, processing others.
pkg_block	Wait until a full message has been read.
pkg_mread	Not used.
pkg_process	Process pkg messages waiting in pkc_inbuf buffer.
pkg_makeconn	Malloc and initialize a pkg_conn structure (connection block).

pkg_dispatch	Send a processed pkg_message to the appropriate user handler.
pkg_gethdr	Get the header off a new message.
pkg_perror	Produces a perror on the error logging output.
pkg_errlog	Default error logger. Writes to standard error.
pkg_ck_debug	Verify debug printing.
pkg_timestamp	Print timestamps to log file.
pkg_suckin	Read all available bytes from the operating system.
pkg_checkin	Check for available input.
pkg_inget	Retrieve a specified number of bytes.

INTENTIONALLY LEFT BLANK.

APPENDIX E:

GENERAL UTILITY ROUTINES

INTENTIONALLY LEFT BLANK.

APPENDIX E-1:

GENERIC LINKED LIST ROUTINES

INTENTIONALLY LEFT BLANK.

"Gen_list.h" and "gen_list.c provide a unified package for the manipulation of linked lists. The lists are composed of elements defined by the structure as shown below.

```
struct list {
    struct list    *li_forw;    /* forward link     */
    struct list    *li_back;    /* backward link    */
    char           *li_ptr;     /* private pointer  */
    long           li_flag;     /* private flag     */
    char           li_type;     /* type             */
}
```

The pointers li_forw and li_back are used for linking the elments together. The pointer li_ptr is used to point to the data portion of the list element (by appropriate casting of pointers). Li_flag and li_type are available for whatever purposes the user deems fitting.

Each list is anchored to a "head" element which, by convention, contains no data. Each element is doubly linked into the list, and the linkage is circular in nature with the list both starting and ending at the "head." This linkage is shown pictorially in Figure E-1.

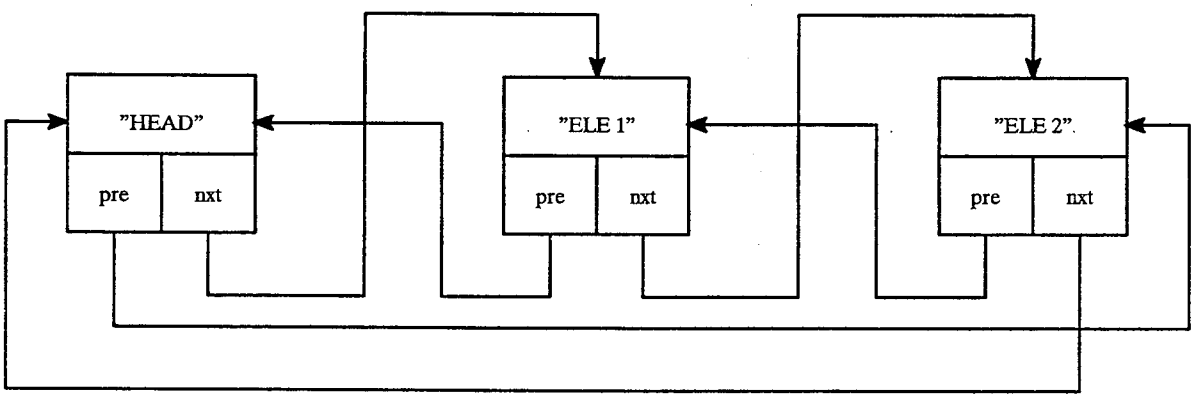


Figure E-1. Generic linked list structure.

A typical loop for traversing such a list is shown below:

```
for:  (lp = l_head.li_forw; lp != &l_head; lp = lp->li_forw) {}
```

Each list is initialized to an empty list configuration as is shown below in Figure E-1-1.

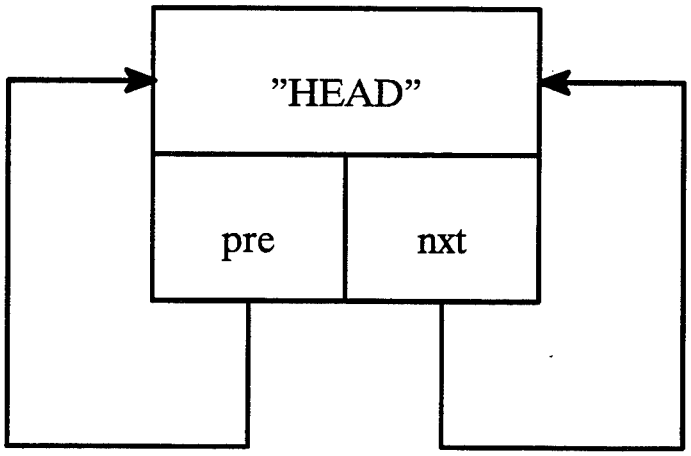


Figure E-1-1. Generic linked list initial configuration.

From this point, a set of macros is used to maintain the list. This set includes the following:

INIT_LIST(cur)	A macro initializing the pointers in a new "head" element appropriately.
GET_LIST(p)	Gets a new list element from a local FreeList. If the last element in this cache has been consumed, the C function, get_list, is called, and another 128 elements are dynamically obtained via malloc.
FREE_LIST(p)	Enqueues a list element back on the FreeList.
INSERT_LIST(new, old)	Insert "new" element in front of "old" list element.
APPEND_LIST(new, old)	Append "new" element after "old" list element.
DEQUEUE_LIST(cur)	Dequeue the element "cur" from the doubly linked list.

APPENDIX E-2:

BIT MANIPULATION ROUTINES

INTENTIONALLY LEFT BLANK.

This appendix contains a number of subroutines for the manipulation of bit strings. The functioning of each routine is documented in the comment statements that precede that routine.

```
/*
 *            B I T   M A N I P U L A T O R S
 *
 *  written by
 *      George Wm Hartwig, Jr.
 *      Aberdeen Proving Ground, MD 21005-5066
 *      February 1991
 *
 *  These routines allow bit fields located at arbitrary positions in
 *  an unsigned character array to be set and read. No error checking
 *  is even thought of, let alone performed. User beware.
 */

/*
 *  Bit-string manipulators for arbitrarily long bit strings
 *  stored as an array of unsigned chars. Bits are assumed to be
 *  numbered from left to right starting with 0.
 */

#define BITCHAR_SHIFT  3      /* log2(sizeof(unsigned char)) */
#define BITCHAR_MASK  0x07 /* base1(BITCHAR_SHIFT) */
#define BITTEST(ip,bit)(ip[bit>>BITCHAR_SHIFT] &
                    (1<<((7-bit)&BITCHAR_MASK)))
#define BITSET(ip,bit)(ip[bit>>BITCHAR_SHIFT] |=
                    (1<<((7-bit)&BITCHAR_MASK)))
#define BITCLR(ip,bit)(ip[bit>>BITCHAR_SHIFT] &=
                     ~(1<<((7-bit)&BITCHAR_MASK)))
#define BIT_MASK 0x1
extern int debug;
/*****************************************************************
 *
 *      W R I T E _ B I T S
 *
 *  written by
 *      George Wm Hartwig, Jr.
 *      Aberdeen Proving Ground, MD 21005-5066
 *      February 1991
 *
 * This function writes "f_len" bits, into the bit stream of unsigned
 * chars "bit_array" beginning at position "f_pos." The value written
 * is taken from the rightmost "f_len" bits of "f_val." Note that a maximum of
 * 32 bits may be contained in f_val.
*****************************************************************/
 write_bits(bit_array, f_pos, f_len, f_val)
```

```
   unsigned char bit_array[] ;
   int f_pos, f_len;
   unsigned int f_val;

{
    if(debug)
        printf("write_bits - fpos = %d, f_len = %d, f_val = 0x%x\n", f_pos,
f_len, f_val);
        while(--f_len >= 0) {

                if(((f_val>>f_len)&BIT_MASK) == 1) {
                    BITSET(bit_array, f_pos);
                }
                f_pos++;
        }
}

/*************************************************************************
 *
 *      R E A D _ B I T S
 *
 *  written by
 *      George Wm Hartwig, Jr.
 *      Aberdeen Proving Ground, MD 21005-5066
 *      February 1991
 *
 * This function reads "f_len" bits, from the bit stream of unsigned chars
 * "bit_array" beginning at position "f_pos."  The result is returned as
 * right-justified bits in an integer (Watch out for fields longer than
 * 31 bits.)
 *
 */

int read_bits(bit_array, f_pos, f_len)
   unsigned char bit_array[];
   int f_pos, f_len;
{

        int byte, byte_pos;
        unsigned int f_val;
       byte = f_len;
        f_val = 0;
        while(f_len-- > 0) {

                f_val <<= 1;
                if(BITTEST(bit_array, f_pos))
                    f_val |= BIT_MASK;
                f_pos++;
```

```
    }
        return(f_val);
}
/*************************************************************************
 *
 *      R E A D _ L O N G _ B I T S
 *
 *  written by
 *      George Wm Hartwig, Jr.
 *      Aberdeen Proving Ground, MD 21005-5066
 *      October 1991
 *
 * This function reads "f_len" bits, from the bit stream of unsigned chars
 * "bit_array" beginning at position "f_pos."  This routine differs in that
 * the field read may be longer than 32 bits.  The result is returned in a
 * dynamically allocated array of unsigned chars.  At some point, the user
 * must free this array.
 *
 */

unsigned char *read_long_bits(bit_array, f_pos, f_len)
   unsigned char bit_array[];
   int f_pos, f_len;
{

    int start;
        unsigned char *f_val;
       start = f_pos;
        f_val = (unsigned char *) malloc(f_len>>3);

        while(f_len-- > 0) {
                if(BITTEST(bit_array, f_pos))
                     BITSET(f_val, f_pos-start);
                f_pos++;
    }
        return(f_val);
}
/*************************************************************************
 *
 *      W R I T E _ B I T S
 *
 *  written by
 *      George Wm Hartwig, Jr.
 *      Aberdeen Proving Ground, MD 21005-5066
 *      February 1991
 *
 * This function writes "f_len" bits, into the bit stream of unsigned
 * chars "bit_array" beginning at position "f_pos". The value written
 * is taken from the rightmost "f_len" bits of "f_cal."
```

```
 *
 */

write_long_bits(bit_array, f_pos, f_len, f_val)
   unsigned char bit_array[] ;
   int f_pos, f_len;
   unsigned char *f_val;

{
     int start;
     printf("write_bits - fpos = %d, f_len = %d, f_val = 0x%x\n", f_pos,
f_len, f_val);
     start = 0;
         while(--f_len >= 0) {
             if(BITTEST(f_val, start) == 1)
                  BITSET(bit_array, f_pos);
             f_pos++;
         }
}
```

APPENDIX E-3:

LINE PARSING ROUTINE

INTENTIONALLY LEFT BLANK.

The routines in this section allow the user to determine a list of delimiters and then proceed to break the furnished string into tokens separated by these delimiters. The result is a pointer to an array of character strings that represents the tokens identified in the original string. The tokens are stored in a static buffer so that the original string is undamaged; however, the user must store the tokens as he sees fit since the next call to parses will destroy tokens generated by a previous call.

```
/*
/*              P A R S E . H
/*
/* A header file for the general purpose scanner parse.c.
/* Anything enclosed within quotes is a string, any contiguous set of
/* characters is a string, and any set of contiguous digits is a string
/* (any imbedded decimal point is assumed to be a part of the string).
/* Leading "0x"s are assumed to be part of the number.
/* Accepted delimiters include
/*  space     tab     null     newline     double quote
/*
*/
char   delims[] = {
    '\\',    ' ',    '\n',    '\t',    '"',    '\0'
};
char **parse();
char **line_parse();
/*
**              P A R S E . C
**
** These functions take a null terminated string which contains a number
** of items separated by delimiters (see parse.h) and returns a pointer to
** an array of pointers which in turn point to the individual items. Each
** item is null terminated in static local buffer, which means that the user
** must copy the items of interest before calling parse again.
**
** Also contained is a binary search algorithm for looking up a string
** from a sorted array of strings.
**
** Line_parse is a routine to put strings back together. The parse routine
** completely tokenizes the input string, which often is not the result
** that was desired.
**
** written by
**     George Wm. Hartwig, Jr.
**
** at the
**     Ballistic Research Laboratory
```

```
**              April 1991
*/
#include "parse.h"

char isdelim[256];   /* hash table of delimiters */
parse_init(){
    int i;
    for(i=0; i<256; i++)
        isdelim[i] = 0;
    i = 0;
    while(delims[i] != '\0')
        isdelim[delims[i++]] = 1;
}

char **parse(s)
register char *s;
{
    static char *bp[128];          /* place to store pointers to tokens  */
    static char  buffer[2048];     /* place to store tokenized string    */
    register char **bpp = bp;      /* pointer to the whole mess          */
    register char *rbp = buffer;
    int i;

    bzero(buffer, 2048);

    for(i=0 ; i<128 ; i++)
        *bpp++ = 0;
    bpp = bp;
    *bpp++ = buffer;
    while(*s > '\0'  && *s != '\n') {
      if(isdelim[*s] == '\0')
       *rbp++ = *s++;
      else {
       if(*(rbp-1) != '\0'){
            *rbp++ = '\0';       /* terminate previous token        */
            *bpp++ = rbp;
       }
       *rbp++ = *s++;            /* include meaningful token in list */
       *rbp++ = '\0';
            *bpp++ = rbp;        /* set the pointer to the next token */
      }
    }
    rbp = 0;
    return(bp);
}
char **sp_parse(s, delim)
register char *s;
char delim;
{
```

```
    static char *bp[1024];          /* place to store pointers to tokens */
    static char  buffer[2048];      /* place to store tokenized string   */
    register char **bpp = bp;       /* pointer to the whole mess         */
    register char *rbp = buffer;
    int i;

    0(buffer, 1024);

    for(i=0 ; i<1024 ; i++)
        bp[i] = 0;
    *bpp++ = buffer;
    while(*s > '\0'  && *s != '\n') {
      if(delim != *s)
       *rbp++ = *s++;
      else {
       if(*(rbp-1) != '\0'){
              *rbp++ = '\0';        /* terminate previous token */
              *bpp++ = rbp;
       }
       *rbp++ = *s++;               /* include meaningful token in list */
       *rbp++ = '\0';
              *bpp++ = rbp;         /* set the pointer to the next token */
      }
    }
    rbp = 0;
    return(bp);
}

/*****************************************************************
 *
 * This routine takes a list of tokens as supplied by the parse routine
 * above and puts double quoted (") strings back together. The quotation
 * marks are lost in the process.
 *
*****************************************************************/

char **line_parse(inptr)
  char *inptr;
{
  register char *s_ptr;
  register char **c_ptr;
  static char *kp_buf[128], **kp_ptr;
  char **parse();
  int i;

      for (i=0; i<128; i++)
       kp_buf[i] = 0;

      kp_ptr = kp_buf;
```

```
    c_ptr = parse(inptr);
  while(*c_ptr != 0) {
    if(**c_ptr == '"') { /* start of string */
            s_ptr = *c_ptr;
            **c_ptr++ = '\0';
            while (*c_ptr != 0){
              if((**c_ptr == '"') && (**(c_ptr-1) != '\\'))
                break;
              if(**c_ptr == '\\'){
                *c_ptr++;
                continue;
              }
              strcat(s_ptr, *c_ptr);
       *c_ptr++;
            }
     *kp_ptr++ = s_ptr;
       **c_ptr++;
    }
     else
       if(**c_ptr != ' '  &&
          **c_ptr != '\0' &&
                **c_ptr != '\t'){
                  *kp_ptr++ = *c_ptr++;
              }
       else
           **c_ptr++;
  }
return(kp_buf);
}
```

LIST OF ABBREVIATIONS

ACISD	-	Advanced Computational and Information Sciences Directorate
ACK	-	Acknowledgment
AFATDS	-	Advanced Field Artillery Tactical Data System
ARL	-	Army Research Laboratory
BFA	-	Battlefield Functional Area
BOM	-	Bit-Oriented Message
BRL	-	Ballistic Research Laboratory
CECOM	-	U.S. Army Communication and Electronics Command
CNR	-	Combat Net Radio
CVC2	-	Combat Vehicle Command and Control
DARPA	-	Defense Advanced Research Projects Agency
DFB	-	Distributed FactBase
DKR	-	Data Kernel Representation
DMD	-	Digital Message Device
FLCS	-	Force-Level Control System
FSK	-	Frequency Shift Keying
IDS	-	Information Distribution System
IDT	-	Information Distribution Technology
LABCOM	-	U.S. Army Laboratory Command
LEC2	-	Lower Echelon Command and Control
MCS	-	Maneuver Control System
NAK	-	Negative Acknowledgment
STScI	-	Space Telescope Science Institute
SECAD	-	Systems Engineering and Concepts Analysis Division
SWS	-	Smart Weapons Systems
TCM	-	Tactical Communication Modem
TOC	-	Tactical Operations Center
TO&E	-	Table of Organization and Equipment
TCP/IP	-	Transport Control Protocol/Internet Protocol

INTENTIONALLY LEFT BLANK.

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR ATTN DTIC DDA DEFENSE TECHNICAL INFO CTR CAMERON STATION ALEXANDRIA VA 22304-6145
1	DIRECTOR ATTN AMSRL OP SD TA US ARMY RESEARCH LAB 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR ATTN AMSRL OP SD TL US ARMY RESEARCH LAB 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR ATTN AMSRL OP SD TP US ARMY RESEARCH LAB 2800 POWDER MILL RD ADELPHI MD 20783-1145
	ABERDEEN PROVING GROUND
5	DIR USARL ATTN AMSRL OP AP L (305)

NO. OF COPIES	ORGANIZATION
1	HQDA USAAIC ATTN LTC WOFFINDEN ARMY 107 PENTAGON WASHINGTON DC 20310-0107
3	DISC4 ATTN DAIS ADO LTC DEAL MAJ BURKE MAJ KINGDON ARMY 107 PENTAGON WASHINGTON DC 2010-0107
1	ARMY DIGITIZATION OFFICE ATTN HQDACS ADO 1745 JEFFERSON DAVIS HWY CRYSTAL SQUARE 4 STE 403 ARLINGTON VA 22202
1	CDR ATTN ATZD CDS MAJ POWERS USAARMC FT KNOX KY 40121
3	CDR ATTN ATZK MW MAJ HERSHEY CPT SPRAGG CPT MACCHIAVELLA USAARMC FT KNOX KY 40120
1	CDR ATTN ATZS CD USAICS FT HUACHUCA AZ 85613-6000
1	CDR ATTN ATZH CDC USASIGCEN FT GORDON GA 30905
1	CDR ATTN ATZH BLT USASIGCEN FT GORDON GA 30905
1	CMDT ATTN ATSF CBL USAFASCH FT SILL OK 73503-5600

NO. OF COPIES	ORGANIZATION
1	PEO CCS ATTN SFAE CC SEO FT MONMOUTH NJ 07703-5207
1	PM AFAS ATTN SFAE CC INT TSE 1616 ANDERSON RD MCLEAN VA 22102-1616
1	PM FATDS ATN SFAE CC FS TMD FT MONMOUTH NJ 07703-5207
1	PM OPTADS ATTN SFAE CC MVR TM FT MONMOUTH NJ 07703-5207
1	DIR ATTN AMSEL RD CS BC CC 2 SALTON CECOM FT MONMOUTH NJ 07703-5207
	ABERDEEN PROVING GROUND
21	DIR USARL ATTN AMSRL-CI-CA, B D BROOME J C DUMER T P HANRATTY R A HELFMAN AMSRL-CI-CC, A E M BRODEEN F S BRUNDICK H CATON S C CHAMBERLAIN A B COOPER III A R DOWNS D A GWYN G W HARTWIG JR R C KASTE M C LOPEZ M J MARKOWSKI C T RETTER L F WRENCHER S D KOTHENBEUTEL AMSRL-CI-CD, J D GANTT AMSRL-SS-IC, P EMMERMAN L TOKARSIK

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-728 Date of Report April 1995

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

_____

_____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.)

_____

_____

_____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

_____

_____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

_____

_____

_____

CURRENT ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001,APG,MD

POSTAGE WILL BE PAID BY ADDRESSEE

DIRECTOR
U.S. ARMY RESEARCH LABORATORY
ATTN: AMSRL-CI-CC
ABERDEEN PROVING GROUND, MD 21005-5067